Some Fundamentals of Meta-Modelling with Application to Measurement of Software Artefact Reuse

by

Eugene Eric Doroshenko, BAppComp(Hons)

Submitted in fulfilment of the requirements

for the degree of

Doctor of Philosophy

Written under the direction of

Professor Glenn R. Lowry

THE UNIVERSITY OF TASMANIA

March, 2005

Statement of Originality

I Eugene Eric Doroshenko (signed hereunder) declare that this Thesis contains no material which has been accepted for a degree or diploma by the University or any other institution, except by way of background information and duly acknowledged in the Thesis, and to the best of my knowledge and belief no material previously published or written by another person except where due acknowledgment is made in the text of the Thesis.

Signed ..... Date 1/6/2005

Eugene Eric Doroshenko, BAppComp(Hons)

Statement of Authority of Access

This thesis may be made available for loan. Copying of this thesis is prohibited for two years from the date this statement was signed; after that time limited copying is permitted in accordance with the Copyright Act 1968

Signed ..... Date 1/6/2005

Eugene Eric Doroshenko, BAppComp(Hons)

Abstract of Thesis

Some Fundamentals of Meta-Modelling with Application to Measurement of Software Artefact Reuse

by

Eugene Eric Doroshenko, BAppComp(Hons)

This thesis proposes a meta-model based framework to investigate measurement of the amount of reuse for a variety of software models. The framework was assessed using a prototype tool.

The evaluation of the framework consisted of four phases that constitute the experiments conducted in the study. The phases are:

- Software Model Type Classification. The framework is assessed to see if it can classify different kinds of software models.
- Software Model Classification. The framework is assessed to see if it can classify and measure size related to the amount of reuse for different software models.
- Measurement Testing. The framework is assessed to see if it can measure the amount of reuse using a series of software models as test cases.
- Automation assessment. The framework is assessed to identify the limits of its application for measurement of the amount of reuse.

Findings from this study indicated that most software model types selected could be measured for the amount of reuse for different software models, provided that each software model type is classified prior to measurement of reuse. This did not require either additional programming or different software components in the prototype tool for different software models. Nor did the framework itself have to be modified at meta-level two, three, or four. Measurement of the amount of reuse using generation provided a reasonably accurate reflection of the actual contribution made using this reuse approach.

Importing of software models using data entry and imported text file did work but was labour intensive. To automate this, tools specific to each software model type need to be made to translate the software model into the text file format defined by the prototype tool. Measurement of reuse using composition for analysis and design models was more realistic in its measurement than for source code models, particularly if the CASE tool used an import/export approach to the reuse of software models. Measurement of the amount of reuse using composition for source code was inaccurate for practical purposes.

Recommendations for further research include the need to streamline the development of tools for translation of software models, refine the framework to include the concept of references for measuring internal reuse using composition, and refine the measurement of reuse for source code and repository-based reuse in CASE technology.

Acknowledgments

I would like to acknowledge all of those who helped in some way to complete this thesis.

Firstly, I would like to thank my supervisor, Professor Glen R Lowry not only for the support he has given me but also for broadening my perspective on University life and its role in society.

There are a number of Academics who helped along the way in this effort. One of them was Felicity Lear, my Honours supervisor who went beyond the call of duty to help me achieve good results in my hours year. The two heads of school I was attached to should definitely be mentioned. They are Professor Chris Keen (Information Systems) and Professor Young Choi (Computing). I would also like to thank Neville Holmes (especially for keeping me one my toes), Bill Morgan, Dr Daniel Rolf, Bob Godfrey (who was my first contact with the University of Tasmania), Paul Crowther (especially for his kindness and assistance during my early undergraduate years), Professor Janet Aisbett, Graeme Shanks, Simon Milton, Dr Alex Rosenberg, and Professor Isaac Balbin. Numerous staff at the office for research provided invaluable assistance and advice. They include Vanessa Folvig, Natasha Petri, Professor Robert Delbourgo, Professor Pip Hamilton, and Jenny Stevenson. A fellow PhD student by the name of Stephen Cahoon deserves a mention for being sociable. A special thanks must go to Professor Leon Sterling and Dr Ed Kazmierczak at The University of Melbourne for their support and advice during the final stages of completion.

A number of general and technical staff also deserve a mention. They are Tony Grey, Kim Strong, Christian McGee, Peter Trueman, Matrin Chung, Cheryl Dinkard, Karen Ward, Chris Purton, Zoe Pearce, and Jane Smith.

A special thanks goes to my mother for being there all this time and Robin Leong for keeping in touch for so long.

I dedicate this thesis to my father, the late Eric Doroshenko.

and my faithful companion, Cerberus.

Preface

The primary purpose of this thesis was to demonstrate how meta-modelling can be used to address a research problem in measurement of software artefact reuse.

The significance of this thesis is twofold. Firstly, this thesis demonstrated that there was a need to provide measurement of software artefact reuse in a consistent way for different modelling paradigms. Secondly, this can be approached through refinement of meta-modelling and application of it to measurement of the amount of reuse.

This thesis is reported in six chapters. A road map of this thesis is illustrated in Figure 1. This is expanded in chapter 4 (Figure 4-1).

Chapter 1 introduces the problem of measurement for software artefact reuse and indicates the possible cause, namely tailoring of software methodologies. The research problem is stated, focusing on how to measure software artefact reuse in a consistent manner suitable for assessment and improvement of software methodology practice.

Chapter 2 is a review of the literature on tailoring of software methodologies, object technology, and measurement of software artefact reuse. The literature review argues a case for the need to tailor software methodologies to different project characteristics. The result is that different modelling paradigms are more or less suitable for different projects. A case is also argued that object technology is no exception to this, even with standards like UML. The literature on measurement of software artefact reuse is then analysed using a model for research surveying to demonstrate that this problem has not been adequately addressed. The chapter concludes with recommendations for research into the measurement of software artefact reuse along with a review of literature that serves as a foundation to addressing the problem in chapter 1.

Chapter 3 describes the specification of a framework using meta-modelling and implemented as a prototype tool (the measurement framework). This is the proposed solution to the problem identified in chapter 2.

Chapter 4 describes the experiments used to test the prototype tool against the hypotheses in the problem statement using a number of experiments (the assessment framework). The assessment framework has four phases:

Phase 1: Software model type classification. This phase tested whether the framework could classify different kinds of software models based on different modelling paradigms.

Phase 2: Software model classification. This phase tested whether the framework could classify different software models based on different modelling paradigms.

Phase 3: Measurement testing. This phase tested whether the framework could measure software artefact reuse for different kinds of software model types and models.

Phase 4: Automation assessment. This phase tested whether the framework was sufficiently generic to address measurement of software artefact reuse based on the previous phases.

Chapter 5 presents analysis of the data obtained using the assessment framework. Analysis was based on hypothesis measures defined in the assessment framework that either support, do not support, or deny a hypothesis. Results are presented for each phase of this thesis and each hypothesis in this thesis. Results for each phase of this thesis are contained in a methodology phase analysis report. This report identifies the values for independent and dependent variables along with the values of hypothesis measures obtained in that phase. Following this is presentation of the results for each hypothesis based on the hypothesis analysis reports. The chapter concludes with an examination of the support for each hypothesis in this thesis.

Chapter 6 presents conclusions and recommendations for further research. This chapter describes the main contributions of this thesis, key findings based on results of this thesis, and issues for discussion and further research related to this thesis.

References, a bibliography, appendices, and a glossary follow. Appendix A and B present details on the measurement framework and the specification of the prototype tool. Appendix C contains some details on the assessment framework and includes a

guide to interpretation of hypothesis measures. Appendix D contains the methodology phase analysis reports and hypothesis analysis reports that include results.

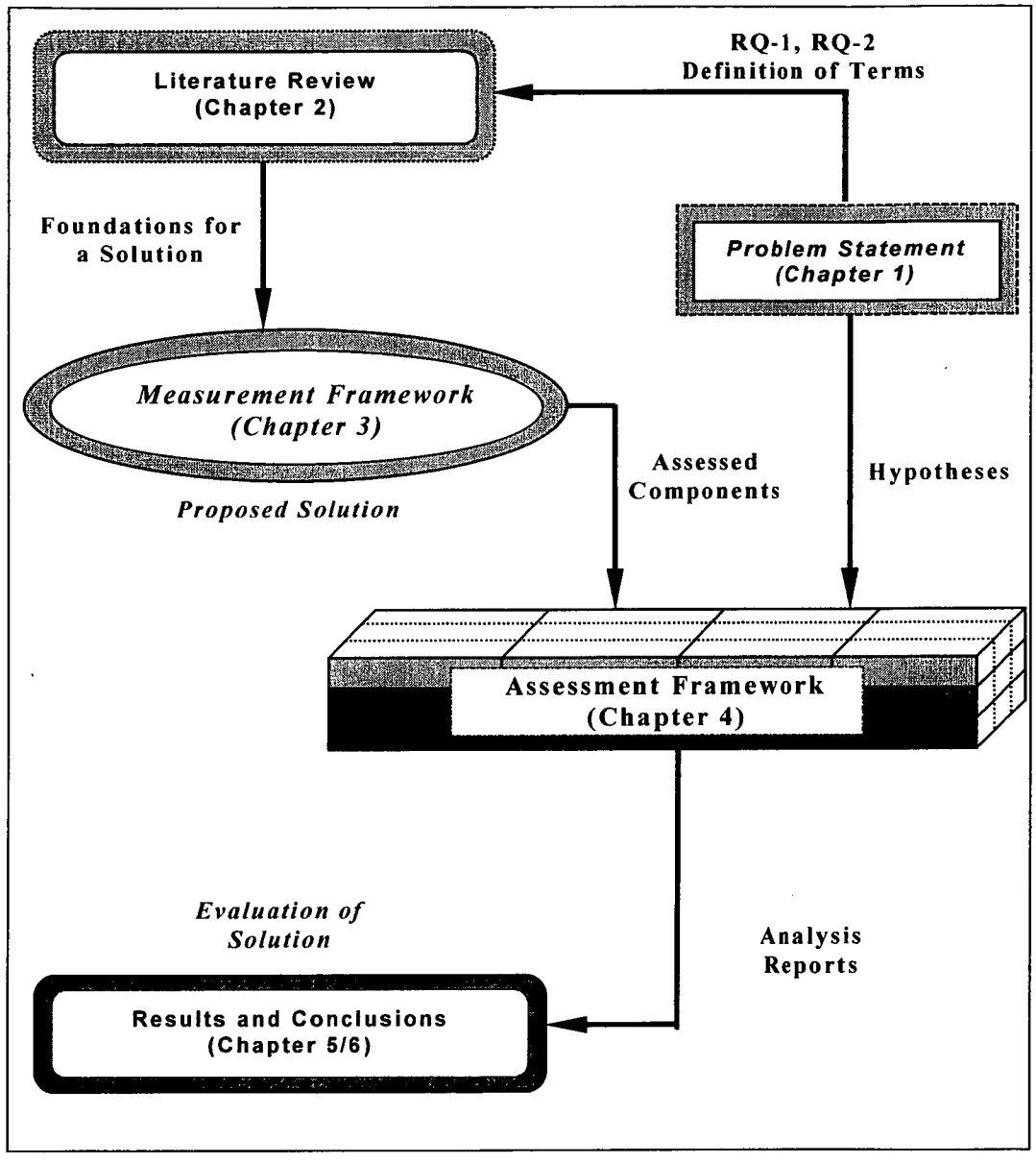

Figure 1: *Thesis Roadmap*

Table of Contents

PREFACE VIII

CHAPTER 1 INTRODUCTION **1**

1.1 PROBLEM STATEMENT 2

1.1.1 Research Questions *2*

1.1.2 Hypotheses *3*

1.2 DEFINITION OF TERMS 3

1.3 ASSUMPTIONS 5

1.4 SCOPE 6

1.4.1 Inclusions *6*

1.4.2 Exclusions *6*

1.4.3 Limitations *7*

1.5 THESIS STRUCTURE 8

1.6 VERIFICATION OF HYPOTHESES 9

Criteria for Validity 9

CHAPTER 2 LITERATURE REVIEW **11**

2.1 CHAPTER OVERVIEW 12

2.2 SOFTWARE REUSE AND REUSE MEASUREMENT 14

The Principle of Reuse: Classification of Reuse Activity *15*

Classifying Reuse by Stage of Development *15*

The Degree of Cognition *16*

Reference Model for Measurement of Software Reuse *17*

Measurement of Generation Reuse *17*

Measurement of Composition Reuse *19*

Measurement of Traceability Reuse *21*

Interpretation of Data *22*

2.3 THE BASIC PROBLEM DEFINED 22

2.4 JUSTIFICATION FOR A GENERAL FRAMEWORK 27

Short History of Software Methodologies *27*

Variation in Object-Oriented and Software Methodologies *30*

Tailoring of Software Methodologies *35*

2.5 MEASUREMENT OF REUSE FOR DIFFERENT SOFTWARE MODEL TYPES 46

A Model for Research Surveying *46*

Research Surveying Tool for the Amount of Reuse *51*

Literature Survey on Measurement of The Amount of Reuse *55*

2.6 TOWARD CONSISTENT MEASUREMENT OF SOFTWARE REUSE 72

Refinement of Meta-modelling in Software Methodologies *72*

Baseline Concepts for Measurement of Reuse *75*

Set Theory for Calculation of the Amount of Reuse *76*

2.7 SUMMARY 79

2.8 SOURCES OF LITERATURE 80

CHAPTER 3 MEASUREMENT FRAMEWORK 82

3.1 CHAPTER OVERVIEW 82

3.2 META-MODEL ARCHITECTURE 83

Fundamentals of Meta-Modelling *84*

Core Concepts 84

Extensions 85

3.3 META-MODEL ARCHITECTURE FOR MEASUREMENT 86

Overview *87*

Static and Dynamic Part *88*

Meta-Model Diagrams *88*

Overview of Layers *91*

M4 Layer *93*

M3 Layer *94*

M2 Layer *96*

M1 Layer *99*

M0 Layer *101*

3.4 OVERVIEW OF MEASURES 102
3.5 DATA CLASSIFICATION 103
Fundamentals *103*
Generation Reuse *105*
Composition Reuse *107*
3.6 SET THEORY 108
Fundamentals *108*
Generation Reuse *110*
Composition Reuse *114*
3.7 MODEL CLASSIFICATION 117
Fundamentals *117*
Representation of Software Model Types *119*
Representation of Software Models *122*
Static and Dynamic Part Revisited *128*
3.8 AUTOMATION SPECIFICATION 129
The Context Data Component *130*
The Model Classification Component *132*
The Measurement Generator Component *133*
Documentation of Specification *135*
Component Specification *136*
Automation Element Specification *136*
Automation Specification Mapping *137*
3.9 SUMMARY 137

CHAPTER 4 ASSESSMENT FRAMEWORK 138

4.1 CHAPTER OVERVIEW 138

4.2 THESIS METHODOLOGY (SYNOPSIS OF THE ARGUMENT) 141

4.3 CRITERIA FOR SUCCESS – CLAIMS MADE 147

4.4 HYPOTHESES AND SUB-HYPOTHESES 148

Sub-Hypotheses for H1 ..... *150*

Sub-Hypotheses for H2 ..... *151*

4.5 DIMENSIONS OF THE ASSESSMENT FRAMEWORK 152

Methodology Phases Dimension *153*

Reuse Approach Dimension *154*

Software Model Types & Implementations Dimension *154*

4.6 METHODOLOGY PHASE SPECIFICATION 156

Requirements for Experiments *156*

Methodology Phase Description Template *157*

Collection Process *159*

Variables *160*

Analysis Process *160*

4.7 THE METHODOLOGY PHASE DESCRIPTIONS 163
Software Model Type Classification: SMTC *163*
Experiment Overview 163
Collection Process 164
Variables 166
Analysis Process 167
Software Model Classification: SMC *177*
Experiment Overview 177
Collection Process 179
Variables 182
Analysis Process 183
Measurement Testing: MT *202*
Experiment Overview 202
Collection Process 204
Variables 208
Analysis Process 209
Automation Assessment: AA *235*
Experiment Overview 235
Defining and Measuring the Concept of a General Framework 238
Collection Process 245
Variables 251
Analysis Process 254
4.8 SUMMARY 277

CHAPTER 5 DATA ANALYSIS AND RESULTS 278

5.1 CHAPTER OVERVIEW 279

5.2 ANALYSIS BY METHODOLOGY PHASE 281

Software Model Type Classification: SMTC 281

Experiment Results and Claim 1 281

Results for Hypotheses and Outcomes 1, 2, and 3 281

Software Model Classification: SMC 285

Experiment Results and Claim 2 285

Results for Hypotheses and Outcomes 1 to 6 285

Measurement Testing: MT 290

Experiment Summary and Claim 3 290

Results for Hypotheses and Outcomes 1 to 6 290

Automation Assessment: AA 299

Experiment Summary and Claim 4 299

Results for Hypotheses and Outcomes 1 to 5 299

5.3 ANALYSIS BY HYPOTHESIS 308

H0a/b 308

H1 312

H2 316

5.4 SUMMARY OF RESULTS 324

CHAPTER 6 CONCLUSIONS **325**

6.1 MAIN CONTRIBUTIONS 326

6.2 KEY FINDINGS 327

6.3 DISCUSSION 329

6.4 FURTHER RESEARCH 333

REFERENCES **340**

CHAPTER 1 340

CHAPTER 2 340

CHAPTER 3 359

CHAPTER 4 360

CHAPTER 5 360

CHAPTER 6 361

LIST OF ABBREVIATIONS **363**

GLOSSARY **365**

APPENDICES **A-1**

APPENDIX A SUPPLEMENTS TO MEASUREMENT FRAMEWORK A-2

Appendix A1 Naming Convention for Sets *A-3*

Appendix A2 Meta-Model Architecture Specification *A-3*

Meta-Model Diagrams A-3

MLC Descriptor A-3

Meta-Model Architecture Diagram of Measurement Framework A-7

M4 Layer A-9

M3 Layer A-15

M2 Layer A-18

M1 Layer A-24

M0 Layer A-35

APPENDIX B AUTOMATION SPECIFICATION B-1
Appendix B1 Component Specification *B-2*
Appendix B2 Automation Element Specification *B-2*
Menu Items B-2
Database Tables B-3
Dialog Boxes B-4
Templates B-4
Classes B-5
Structs B-7
Functions B-7
Appendix B3 Automation Specification Mapping *B-8*
MLC to Menu Item B-8
MLC to Database Table B-8
MLC to Dialog Box B-9
MLC to Template B-10
MLC to Class B-10
MLC to Struct B-11
MLC to Function B-12

APPENDIX C SUPPLEMENTS TO ASSESSMENT FRAMEWORK C-1

Appendix C1 Methodology Phase Description Template *C-2*

Collection Process C-4

Variables C-6

Analysis Process C-7

Appendix C2 Guide to Interpretation: Software Model Type Classification Methodology Phase *C-21*

Hypothesis Measures C-21

Assessment Criteria C-32

Appendix C3 Guide to Interpretation: Software Model Classification Methodology Phase *C-47*

Hypothesis Measures C-47

Assessment Criteria C-69

Appendix C4 Guide to Interpretation: Measurement Testing Methodology Phase *C-94*

Hypothesis Measures C-94

Assessment Criteria C-151

Appendix C5 Guide to Interpretation: Automation Assessment Methodology Phase *C-243*

Hypothesis Measures C-244

Assessment Criteria C-334

Appendix C6 Format of Analysis Reports *C-423*

Format of Methodology Phase Analysis Report C-423

Format of Hypothesis Analysis Report C-424

APPENDIX D SUPPLEMENTS TO DATA ANALYSIS D-1

Appendix D1 Analysis Report for Software Model Type Classification *D-2*

Appendix D2 Analysis Report for Software Model Classification *D-6*

Appendix D3 Analysis Report for Measurement Testing *D-17*

Appendix D4 Analysis Report for Automation Assessment *D-36*

Appendix D5 Analysis Report for Hypothesis H0 ..... *D-79*

Appendix D6 Analysis Report for Hypothesis H1 (All Software Model Types) *D-88*

Appendix D7 Analysis Report for Hypothesis H1 (OO/UML Software Model Types) *D-94*

Appendix D8 Analysis Report for Hypothesis H2 (All Software Model Types) *D-100*

Appendix D9 Analysis Report for Hypothesis H2 (OO/UML Software Model Types) *D-128*

Appendix D10 Software Model Types and Implementations Used *D-156*

Software Model Types Table D-156

Implementations Table D-162

Software Model Types and Implementations Table D-163

Appendix D11 Experiment Summaries *D-165*

Software Model Type Classification Experiment Summary D-165

Software Model Classification Experiment Summary D-166

Measurement Testing Experiment Summary D-170

Automation Assessment Experiment Summary D-173

Chapter 1 Introduction

Software reuse, particularly requirements reuse, has gained renewed interest in the 1990s with growth in demand for distributed software systems, telecommunications software, and software for electronic commerce and integrated manufacturing. As a result, the object-oriented paradigm for software development has emerged as a major candidate for supporting reuse. Object technology has matured from its beginnings as a subject for individual researchers, to a major academic research area with conferences and books devoted to the subject; to standards for modelling, interoperability, and portability. Two major *de facto* standards have emerged for object technology: these are the Unified Modelling Language (UML) for modelling and Interface Definition Language (IDL) in the Object Management Group (OMG) standards for interoperability and portability. Measurement of the amount of reuse has emerged as one key measure to assist evaluation of reuse activity, the effect of reuse on project goals, and identification of software assets.

Considerable variety in object technology does exist and both the UML and IDL standards were introduced to reduce the problems caused by real and apparent differences in vocabulary and notation. Is the variation due just to basic differences in a methodologist's perspective or is there a more fundamental cause? Some researchers have described one possible cause of variation as the need to tailor software methodologies to suit problem domains, projects, organisations, or any combination of these. To cope with this, researchers have responded by forming various frameworks and models, and Computer Aided Software Engineering (CASE) tool vendors have responded by building meta-CASE tools. Both the frameworks and tools are designed to tailor software methodologies to projects, problem domains, and organisations. This includes changing or adapting the kinds of software models used. Thus, what may have been avoided with UML and IDL has perhaps re-emerged as a problem for measurement in software methodologies, including the amount of reuse. Some researchers in software methodology areas, including tailoring, reuse, and measurement, also recognise the effect of the problem domain on software development success. Models for tailoring software methodologies aim to institutionalise expertise in the problem domain. Research in reuse aims to capture the reusable artefacts of the problem domain.

The impact on measurement is twofold. Firstly, a change in the problem domain and other factors (organisation, project, etc) affects the values obtained. Secondly, the kind of software model or modelling paradigm affects how the values are calculated. If practitioners wish to measure the amount of reuse for new modelling paradigms or

their own extensions to them, they must either wait for researchers to develop measures or they must do it themselves. Even with the use of IDL and UML, analysis of data still needs to support evaluation of reuse activity. Moreover, little research has been done to address measurement of the amount of reuse based on UML and IDL. *Formation* of measures is a research activity. This makes it time consuming and any number of options for measurement can be obtained, leading to potential inconsistency for evaluation of reuse activity using different paradigms.

Rather than invent measures from scratch for each new modelling paradigm, perhaps a better solution is to support a process for deriving and applying measures of the amount of reuse using some general framework. In this way the problem of invention is minimised to the essential requirements of the measure and, as software model types are modified or invented, measurement of the amount of reuse is more consistent and more readily available. The framework could be along the lines of meta-CASE tools, tailoring of software methodologies to problem domains, and software measurement where meta-modelling is prominent. This framework could be tailored to measure the amount of reuse for different kinds of software models. The framework could be evaluated using a selection of object-oriented, UML, and other kinds of software models.

1.1 Problem Statement

This study aims to evaluate the need for and capability of a framework for different kinds of software models that supports measurement of the amount of software reuse. A framework based on meta-modelling is proposed such that the elements of specific models are classified, transformed into sets, and the amount of reuse measured. This thesis evaluates the framework using relevant literature, a prototype tool, and a selection of automated software models. The proposed framework is general; the aim is to measure the amount of reuse for numerous kinds of software models. For example, the framework should easily cope with UML, E-R models, and Data Flow Models.

1.1.1 Research Questions

RQ-1: Does the proposed framework support the measurement of the amount of reuse for different kinds of software models[1]?

RQ-2: What limitations does the proposed framework have for measurement of the amount of reuse for different kinds of software models?

[1] Different kinds of software models includes UML and object-oriented software models.

1.1.2 Hypotheses

The hypotheses are established to investigate the research questions. The Null hypothesis is H0, the remaining hypotheses are research hypotheses.

H0a: A framework based on meta-modelling cannot support measurement of the amount of reuse for any kind of software model.[2]

H0b: A framework based on meta-modelling does not have any limitations in measurement of the amount of reuse with different kinds of software models.

H1: A framework based on meta-modelling can support measurement of the amount of reuse for different kinds of software models.

H2: A framework based on meta-modelling has limitations in measurement of the amount of reuse with different kinds of software models.

1.2 Definition of Terms

In formulating the hypotheses and research questions above the following terms are used which are defined below.

T1: **Limitation:** Limitation is measured in the following ways in this thesis:

- Each time we have a different classification for each new software model type or each new software model then the framework is limited.
- If the accuracy of reuse measurement was poor then the framework is limited.
- If changes were made to the static part of the meta-model architecture during the experiments then the framework is limited.
- If changes were made to the dynamic part of the meta-model architecture during the experiments then the framework is limited.

T2: **Software model:** A model that relates to the functionality of software relative to some level of abstraction. For example, an analysis model or design model.

[2] In this thesis the range of possible software model types is limited to those identified in the "Software Model Types Table", Appendix D10, pages D-156 to D-159.

T3: **Kind of software model:** The foundation upon which a software model is based. For example, an analysis model can be a state chart, petri net, or data flow model. In this thesis the range of possible software model types is limited to those identified in the "Software Model Types Table", Appendix D10, pages D-156 to D-159.

T4: **Software model type:** A formal term for a kind of software model.

T5: **Meta-modelling:** Use of a model to describe and define the structure of other models.

T6: **UML:** Unified Modelling Language. A de facto standard for object-oriented modelling.

T7: **UML software model:** A software model type that is part of UML.

T8: **Object-oriented software model:** A software model type based on the object-oriented paradigm.

T9: **Modelling paradigm:** The basis or foundation for a software model type. For example, a class model is a software model type based on the object-oriented paradigm. The modelling paradigm is the object-oriented paradigm.

T10: **Software process:** A description of the procedures and documentation by which software is developed.

T11: **Software development:** Development of software based on a software process.

T12: **Process of measurement:** The procedures followed when applying measurement to evaluate the software process.

T13: **Historical data:** Measurements gathered during software development and used to evaluate the software process.

T14: **Software reuse:** Reuse of one artefact in another artefact that is a product of software development.

T15: **Amount of reuse:** The extent to which an artefact contributes to the content of another artefact.

T16: **Reuse activity:** The activity through which reuse is done or performed.

T17: **Set theory:** The properties of sets found in mathematics, particularly discrete mathematics.

1.3 Assumptions

These are the assumptions related to the study.

AN-1: The study assumes that reuse is not discouraged in software development organisations.

AN-2: The study assumes that the means for assessing reuse benefit are available.

AN-3: The study assumes that all software projects and products produced as a result of the software projects are for a single organisation.

AN-4: The study assumes that experts, when compared to novices, have higher values for amount generated (ag) and amount reused (ar), and lower values for amount added (aa), amount not generated (ang), waste generated (wg), and amount not reused (anr).

AN-5: The study assumes that a framework based on meta-modelling can support analysis of historical data for reuse activity for different kinds of software models.

AN-6: The study assumes that a framework based on meta-modelling can support the formation, collection, analysis and interpretation stages of a process of measurement for the amount of reuse for different kinds of software models.

AN-7: The study assumes that a framework based on meta-modelling can support analysis of historical data for reuse activity for object-oriented and UML software models.

AN-8: The study assumes that a framework based on meta-modelling can support the formation, collection, analysis and interpretation stages of a process of measurement for the amount of reuse for object-oriented and UML software models.

1.4 Scope

In considering the scope of the study in this thesis the following inclusions and exclusions are made.

1.4.1 Inclusions

IN-1: The study explored the effectiveness of automating measurement for the amount of reuse.

IN-2: The study explored the fundamentals of meta-modelling using measurement of the amount of software reuse with automation.

1.4.2 Exclusions

EX-1: The study does not explore the effectiveness or practice of software reuse in organisations.

EX-2: The study did not deal with or address the management issues and perceptions of software reuse, especially in relation to productivity.

EX-3: The study did not explore the representation issues of object-oriented software models, or any other kind of software model.

EX-4: The study did not explore the user interface issues of CASE tool construction for prototyping of software.

EX-5: The study did not discuss performance of industrial CASE tools.

EX-6: The study did not explore measurement of the amount of reuse for kinds of software models that are not automated.

EX-7: The study did not explore the representation issues of meta-models or meta-modelling.

EX-8: The study did not explore the performance issues related to CASE technology that supports measurement of the amount of reuse.

EX-9: The study did not assess the reuse benefit of different kinds of software models.

EX-10: The study did not explore identification of software assets for different kinds of software models.

EX-11: The study did not gather data about software reuse from organisations, such as source code or software models, to test the measurement framework in chapter 3.

EX-12: The study did not explore the effectiveness of meta-modelling for analysis of reuse activity or reuse benefit using different kinds of software models.

EX-13: The study did not explore process support for measurement of the amount of reuse for different kinds of software models using meta-modelling.

EX-14: The study did not explore automation support for measurement of the amount of software reuse for evaluation of reuse activity.

EX-15: The study did not explore automation support for a process of measurement related to measurement of the amount of software reuse.

1.4.3 Limitations

LN-1: The study was limited to Borland C++ 5.02 Developer Suite on Windows 95 for construction of any prototype tool.

LN-2: The study was limited to measurement of the amount of reuse by comparing the original software artefact with the final version of another software artefact that reused it.

LN-3: The study was limited to ownership of a software model for the purposes of determining internal or external reuse.

1.5 Thesis Structure

Figure 1 illustrates the relationship between the Problem Statement (Chapter 1), the literature review (Chapter 2), the measurement framework (Chapter 3), the assessment framework (Chapter 4), and the results after execution of the assessment framework (Chapter 5), and conclusions drawn from the results (Chapter 6). To summarise:

- The *problem statement* defines the problem. Basically it is proposed that measures for the amount of reuse should be defined using a general framework for a range of software model types to ensure that these measures are consistent.
- The *literature review* verifies that the problem exists and provides a foundation for the *measurement framework*. There is evidence to suggest that different kinds of software models will continue to exist to specify different kinds of systems. There is no way to measure the amount of reuse in a consistent way for the various kinds of software models.
- The *measurement framework* is the proposed solution to the problem. The measurement framework is designed to measure the amount of reuse in a consistent way for different kinds of software models, including future software model types.
- The *assessment framework* evaluates the *measurement framework* based on the problem statement. The *assessment framework* represents the experiments used to evaluate the proposed solution to the problem. The assessment framework is designed to verify that the measurement framework can measure the amount of reuse for different kinds of software models.
- The *analysis reports* from the *assessment framework* are the results of the experiments.
- The *results and conclusions* evaluate the proposed solution based on results.

Note that this thesis is concerned with the formulation of measures for the amount of reuse for different kinds of software models, that is, finding a consistent way to measure reuse with different kinds of software models. This thesis is *not* concerned with acceptance or perceptions of reuse practice in organisations nor is it a study designed to gather data about reuse practice. To gather data about reuse practice and make meaningful comparisons requires consistent measures for different kinds of software models. The point of this thesis is to test a framework for measurement of reuse prior to gathering data about software reuse. Readers should pay particular attention to exclusions EX-1 and EX-11.

1.6 Verification of Hypotheses

This section summarises the steps taken to verify the hypotheses.

Step 1 Derive a meta-model based framework for measuring the amount of reuse.

Step 2 Define an experiment to see if the measurement framework can classify different software model types and different software models. Both popular and odd software model types are selected. The experiments for this are software model type classification and software model classification. See chapter 4, section 4.7, under the headings "Software Model Type Classification" and "Software Model Classification" for further details.

Step 3 Define an experiment to see if the measurement framework can measure the amount of reuse accurately. The experiment for this is measurement testing. See chapter 4, section 4.7, under the heading "Measurement Testing" for further details.

Step 4 Define an experiment to see if the measurement framework can measure the amount of reuse without changes to the static part of the framework. The experiment for this is automation assessment. See chapter 4, section 4.7, under the heading "Automation Assessment" for further details.

Step 5 Measure the amount of reuse without changes to the dynamic part of the framework. The experiment for this is automation assessment. See chapter 4, section 4.7, under the heading "Automation Assessment" for further details.

Criteria for Validity

The data gathered reflects the standard ways of measuring reuse in the literature (See section 2.5). The nature of the data does not lend itself to analysis based on variance because for any given experiment, the measurement framework either works or does not work for a given software model type. For example, out of the total number of software model types classified, how many of them could the measurement framework measure the amount of reuse.

For Step 2, the Percentage of software model types and software models classified indicates the degree of support for H1. The Percentage of software model types and software models unclassified indicates the degree of support for H2.

For Step 3, for any software model type the amount of reuse measured must be correct for all test cases, where each test case has a known amount of reuse in a software model. This supports H1. If any test case fails this indicates support for H2.

For Step 4, changes in the static part of the framework is measured by changes in the software tool that implements the framework. Any changes in the static part supports H2 and does not support H1.

For Step 5, changes in the dynamic part of the framework is measured by changes in the data maintained by the software tool that implements the measurement framework. Any changes in the dynamic part of the measurement framework supports H2 and does not support H1.

Chapter 2 Literature Review

Software reuse is the redeployment of artefacts from software development. Reuse can occur where a software artefact is reused as a product (composition reuse), or used to generate another artefact (generation reuse). Internal reuse is the reuse of artefacts from the same project. External reuse is the reuse of artefacts outside a project. The amount of reuse is a measure of how much an artefact contributes to the development of another artefact. Together with the stage of development, a number of possibilities exist to measure the amount of reuse based on composition or generation reuse and internal or external reuse.

The culmination of research into software methodologies is object-oriented methodologies with de facto standards such as UML and the Object Management Group standards body. Object-oriented methodologies were created to address the limitations of previous methodologies for software development. The OO paradigm has led to a number of variations as well as efforts to provide a coherent view of OO through publication of various taxonomies and standards.

Although standards for object technology like UML could streamline the measurement of reuse, there are still variations in object-oriented methodologies as well as variations in software methodologies in general. One issue that needs further exploration is the cause of variation in object-oriented methodologies and what the implications are for measurement of the amount of reuse for software artefacts. This is the main subject of the literature review.

We will begin by giving an overview of problems encountered when measuring reuse of software model types. Then an instrument is created to survey existing work in measurement of the amount of reuse. The instrument will allow us to systematically review and assess previously published work and is summarised in Tables Table 2-4 to Table 2-13 in section 2.5.

2.1 Chapter Overview

The literature review seeks to introduce some basic concepts related to software reuse and reuse measurement and present evidence on the need for a framework to measure the amount of reuse for different kinds of software models. Following this is brief justification is given for the use of meta-modelling to define the framework. The literature review has 7 main parts. These are:

2.2 ***Software Reuse and Reuse Measurement*** explains the key concepts and terms used in this thesis for software reuse and measurement of software reuse.

2.3 ***The Basic Problem Defined*** describes in simple terms the problem that is faced for measurement of the amount of reuse. This can be summarised as follows. Software methodologies use various kinds of software models. This is essential to improve outcomes for software projects. Different kinds of software models require measures that are suited to each kind of software model and yet these measures must be consistent to make comparisons of software methodologies that use different kinds of software models. This is the problem that has not been adequately resolved.

2.4 ***Justification for a General Framework*** reinforces the need for a general framework by drawing on literature that illustrate variation of software model types is an essential feature in software methodologies and we therefore need to provide a framework that can measure the amount of reuse for different kinds of software models. This is done from three perspectives, namely, the history of software methodologies, the variation in object-oriented methodologies, and tailoring of software methodologies.

> ***Short History of Software Methodologies*** describes how variation of software methodologies have always existed and this includes variation of software model types. The evidence suggests that this was done to improve the outcomes of software projects. New kinds of projects were encountered and this necessitated the need for new kinds of software models to ensure a complete and consistent specification. The lesson of history is that different kinds of software models have existed in the past and this is likely to happen in the future.
>
> ***Variation in Object-Oriented and Software Methodologies*** illustrates that variation in object-oriented methodologies also implies variation in the kinds of software models. In addition, sources are included to illustrate that since object-oriented methodologies came into existence other kinds of software model types have been refined or developed. UML is also addressed to illustrate that it also has a range of software model types and is constantly being improved to account for different kinds of software models. The lesson

here builds on what was observed in the past history of software methodologies. Variation in software methodologies and software model types occurred in past and continues to exist in the present; this is evident in the recent literature.

Tailoring in Software Methodologies goes one step further to define the reason why variation in software methodologies and software model types is essential. The Basic reason is that different kinds of software systems have different kinds of features that need to be modelled using different kinds of software models to ensure a complete and consistent specification. If a screw is required then we use a screw with a screw driver, not a hammer and nail even though it may be cheaper. If we do this then the outcomes of software projects are improved. The lesson here is most profound. We needed different kinds of software models to ensure adequate outcomes for software projects in the past. This has also occurred in the present. This gives us a reason to believe that variation in software model types will occur in the future. The implication for reuse is that we are going to reuse different kinds of software models to reduce development time and further improve outcomes by decreasing defect rates. The implication for reuse measurement is that we need to measure reuse for different kinds of software models to detect reuse activity and its impact on outcomes for projects.

2.5 ***Measurement of Reuse for Different Software Model Types*** evaluates the various approaches to measurement of reuse to illustrate that work in measurement of reuse does not adequately address the need to consistently measure reuse for different kinds of software models. Although many measurement models were found, most of them were developed for a specific kind of software model, and most of these were implementation (source code). Few models exist for analysis and design, and no model was found that was applied to a range of software model types to provide a consistent set of measures for evaluation of different software methodologies.

Findings indicate that it is possible to find measures for any given software model type, but a common set of measures for a range of software model types was not found. If we use two different measures for reuse for two kinds of software models this would be like comparing volume for one car to the weight of another car to see if we made a better car using different manufacturing techniques. We cannot draw any reasonable conclusions because the contrast in what is being measured is so vastly different from one software model type to another, and this is the case with work done in measuring the amount of reuse. Few sources addressed measurement of software reuse at analysis and design stages and only X were found that demonstrated their measurement theory using particular kinds of software models.

2.6 Toward Consistent Measurement of Software Reuse proposes meta-modelling as a foundation for the structure of a general framework because of its prominent use in tailoring of software methodologies to address the need for specification of different kinds of software models with automation. The reason for this is best described as a rhetorical question. A meta-case tool automates the modelling process for different kinds of software models so why not make a meta-case tool to automate measurement of the amount of reuse for different kinds of software models? In addition, concepts from set theory and reuse measurement are also recommended as a foundation for framework content.

2.2 Software Reuse and Reuse Measurement

Software reuse can be traced back to the sixties with the introduction of machine code generation reuse using structured text specifications. These tools were called compilers. Ghezzi et al [1] give numerous examples of more traditional reuse including personnel, operating systems, database engines, as well as libraries of source code.

Many artefacts or products arising from the development of software are candidates for reuse, including people, test cases, analysis models, design models, project plans, source code, and development processes [1-15].

Layzell and Freeman [3] distinguish between three kinds of sources for reuse: dynamic knowledge sources (human experts), informal knowledge sources (documentation and semi-structured representations), and formal knowledge sources (source code and structured representations). This thesis deals with software models as formal knowledge sources, particularly those that are automated.

Muhanna [16] distinguishes between model types for defining models in decision support systems. Similarly, software models in software methodologies are treated as instances of software model types. For example, a class model for a weather monitoring station is an instance of the model type named class model. Software model types are also composed of model element types. For example, a class model contains associations and classes.

The following concepts are based on the reference models in two sources [8, 17], and is centred primarily around software models. The model is refined in this thesis to account for measurement of the amount of reuse (Figure 2-1, Figure 2-2, Figure 2-3). Three dimensions can be identified that govern reuse of software models. These are the stage of development, the degree of cognition, and the principle.

The Principle of Reuse: Classification of Reuse Activity

Two principles for reusing a software model are generation and composition. Composition reuse refers to software models as products that are reused in other software models. An example is the reuse of source code libraries in C++ using the #include directive, or importing of previous class diagrams in current diagrams. For composition reuse, the work distinguishes between the library model and the application model. The library model is reused through composition (for example, the C++ libraries). The application model reuses the library model through composition (for example, the C++ application). Generation reuse refers to software models are used to generate other software models, some of which can be executable. For example, generation of C++ source code from a Booch [11] class diagram.

For generation reuse, this work distinguishes between the source model, generated model, and complete model. The source model is reused through generation (for example, the class diagram). The generated model is generated from the source model (for example, C++ code generated from the class diagram). The complete model reuses the source model through generation (for example, a C++ application). Composition reuses a software model as a product. Generation reuses the transformation knowledge for a software model.

Fenton [18] and Frakes and Terry [12] propose two variations for composition reuse. Public or external reuse is where software models in a library or from old projects are reused in new projects. An example is reuse of a queue class in a robot arm class. Private or internal reuse is where parts of software models are reused within the same project. For example, reuse of an error handling class as an instance variable in other classes. External reuse using generation occurs where a software model from a library or previous project is used to generate another software model in a current project. Internal reuse for generation occurs within the same project where a software model created in a project is used to generate another software model in the same project. For generation, internal reuse is more common than external reuse. However, generation reuse is one foundation for reuse using traceability described below under the heading "Measurement of Traceability Reuse".

Classifying Reuse by Stage of Development

A software model is produced at a given stage of development. The three significant stages of development are analysis, design, and implementation. Each of these refers to the stage or phase of development for a software model. For example, reuse of a data flow model is analysis reuse. Reuse of source code is implementation reuse. All this implies that the software model can vary with the stage of development and the methodology. For example, in structured analysis we reuse data flow models, in

structured implementation COBOL is reused; in object-oriented analysis we reuse use case models, in object-oriented implementation C++ may be reused. Further elaboration on the kinds of stages and their synonyms can be found in [19].

The Degree of Cognition

The degree of cognition refers to the ability of a developer to reuse software models. This can be either reasoning or general problem solving, analogical reasoning or semi-expert problem solving, or expert problem solving [17, 20, 21]. A developer relies on general problem solving skills if they encounter a problem in a domain in which they have no significant knowledge. A developer relies on semi-expert problem solving skills if they encounter a problem in a domain that is similar to a domain in which they have significant knowledge for solving it.

A developer relies on expert problem solving skills if they encounter a problem in a domain that is the same as the one in which they have significant knowledge. As developers gain experience in modelling in a given problem domain, they become more expert and better able to identify reuse opportunities. More experienced developers are more effective at reusing software models than less experienced ones. Expertise is based on the experience of a developer in a given problem domain [20, 21].

The degree of cognition can be institutionalised to some degree through automation. This is usually done by capturing expertise attached to a problem domain [21]. In composition reuse, expertise of a problem domain is manifest in artefacts that model it. For example, reuse of an ADT queue in C++ source code can be considered composition reuse with general problem solving. This is sometimes referred to as horizontal or application-wide reuse. Reuse of an Accounts Payable subsystem in a class diagram can be considered composition reuse with expert problem solving. This is sometimes referred to as vertical or domain specific reuse. In generation reuse this is manifest in artefacts that are modelled using some software model type specifically tailored to a problem domain. For example, generation of C++ code from a class diagram can be considered automation of generation reuse with general problem solving. Generation of C++ code using an application generator for real-time systems can be considered generation reuse with expert problem solving. Two indicators of the value of expertise in reuse are the vertical domain interfaces in OMG standards for composition reuse [22], and the construction of application generators for generation reuse [8, 13, 17].

Reference Model for Measurement of Software Reuse

Frakes and Terry [12] survey a number of models and metrics for software reuse. This work focuses primarily on gauging the amount of reuse. Amount of reuse refers to the measurement of how much a particular software model is reused in other software models. The measure has three basic uses: assessment of reuse activity, evaluation of reuse benefit, and identification of software assets.

Reuse activity is manifest in software artefacts. Software artefacts that are reused indicate development with reuse. Software artefacts that are made and reused indicate development for reuse. Measurement of the amount of reuse gives a value for this activity occurring. To summarise, the amount of reuse can help answer the question "Are we practising reuse?". In this way an official process is evaluated against the actual process.

To assess the benefits of reuse, values for the amount of reuse are correlated with performance indicators such as productivity, defect rate, and user satisfaction. In this way, measurement of the amount of reuse also helps answer another question, namely "Does reuse make a difference?". Given that reuse does yield benefits, ascertaining the amount of reuse can help identify software assets based on reuse and helps answer the question "Where are the software assets that improved performance?".

The reference model for measurement of amount of reuse classifies the various approaches to reuse and their relationship to reuse amount (See Figure 2-1, Figure 2-2, and Figure 2-3). The amount of reuse is classified as an internal product attribute measure with formation, collection, analysis and interpretation stages.

Measurement of Generation Reuse

Internal reuse using generation occurs where a software model (the generated model) is generated from another software model (the source model), usually produced in an earlier stage of development. The generated model then appears as part of the final version of the software model (the complete model) for the stage of development. The source and complete models are part of the same project. The amount of reuse is calculated by comparing the generated model to the complete model. For example, in Figure 2-1a) the class model named Vehicle Inventory (the source model) is used to generate part of the C++ model named VehInv Application (the complete model). The model generated from the Vehicle Inventory model (the VehInv Generated model) is compared to the VehInv Application model to calculate the amount of reuse.

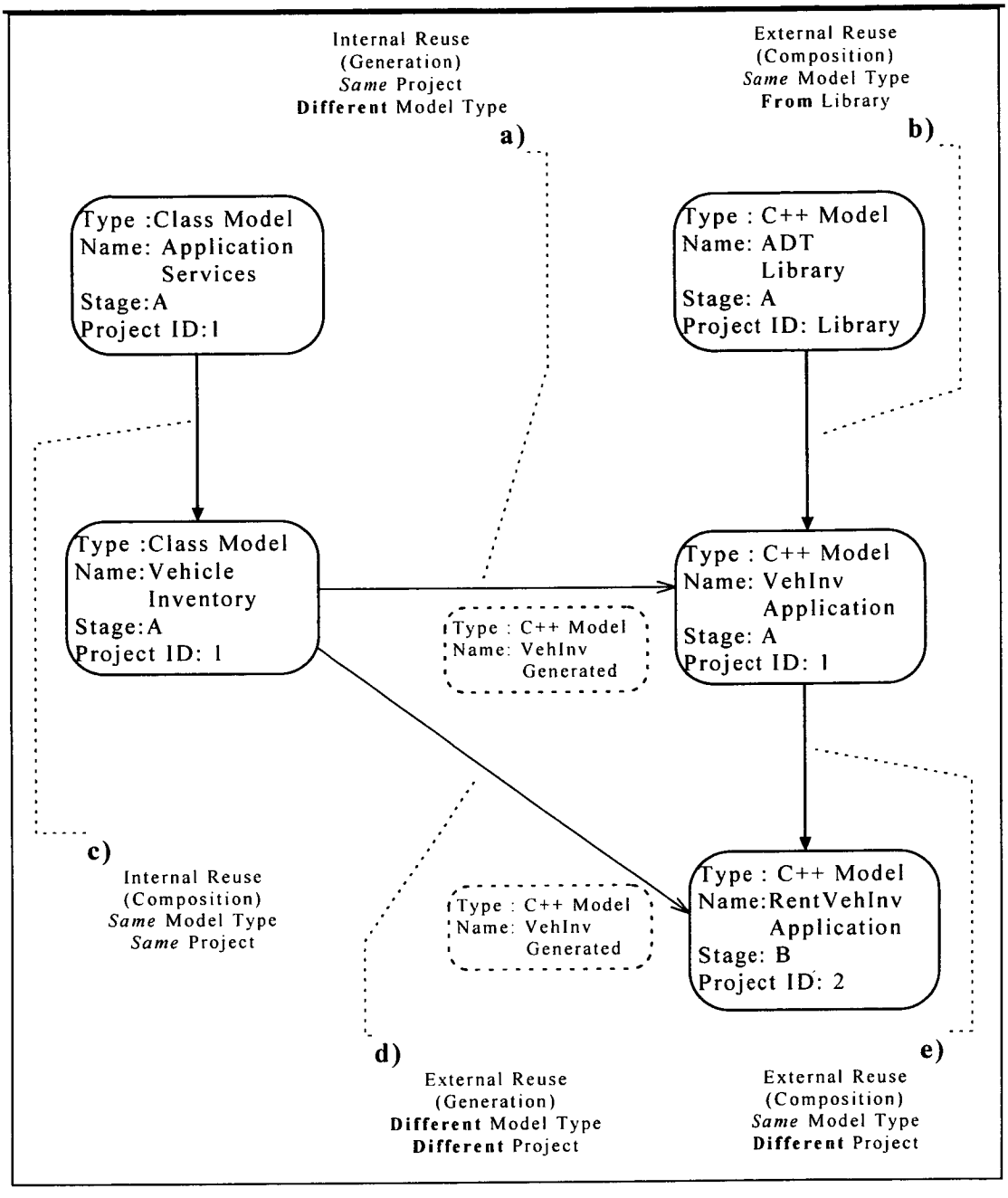

Figure 2-1: *Internal and external reuse based on definitions*

External reuse using generation occurs where a software model (the generated model) is generated from another software model (the source model). The generated model then appears as part of the final version of the software model (the complete model) for the stage of development. The source model is part of a previous project or library of software models. The amount of reuse is calculated by comparing the generated model to the complete model. For example in Figure 2-1d) the class model named Vehicle Inventory (the source model) is used to generate part of the C++ model named RentVehInv Application (the complete model). The model generated from the VehInv Application (the VehInv Generated model) is compared to the model named RentVehInv Application to calculate the amount of reuse.

Using [23] as an example, a_i can be the amount generated in terms of components generated and used in a complete model, and A_i can be the number of components that make up a complete model. Thus,

$$\text{The amount of reuse} = (a_i \div A_i) \times 100$$

So a_i is the number of components in the VehInv Generated model that are also in the RentVehInv Application model, and A_i is the number of components in the RentVehInv Application model.

Measurement of Composition Reuse

External reuse using composition occurs where a software model from a different project or library (the library model) is reused in another software model (the application model). The software models are usually of the same software model type. The amount of reuse is calculated by comparing the library model with the application model. For example, in Figure 2-1b) the C++ model named ADT Library (the library model) is reused in the C++ model named VehInv Application (the application model). The amount of reuse is calculated by comparing the ADT Library model to the VehInv Application model. This is an example of external reuse using a library. In Figure 2-1e) the C++ model named VehInv Application (the library model) is reused in the C++ model named RentVehInv Application (the application model). The amount of reuse is calculated by comparing the VehInv Application model to the RentVehInv Application model. This is an example of external reuse using another project.

A basic example of a measure can be found [12]. The calculation is

$$\text{amount of reuse} = \frac{\text{amount reused in application model}}{\text{size of the application model}}$$

So, this gives us the following calculation for VehInv Application model and RentVehInv Application model.

$$\text{amount of reuse} = \frac{\text{amount reused in RentVehInv Application model}}{\text{size RentVehInv Application model}}$$

Internal reuse using composition has two approaches.

- Where a software model (the library model) is built and reused in another software model (the application model) in the same project, the amount of reuse is calculated by comparing the library model with the application model. For example, in Figure 2-1c) the class model named Application Services (the library model) is reused in the class model named Vehicle Inventory (the Application model). The amount of reuse is calculated by comparing the Application Services model with the Vehicle Inventory model.

- Where a model element is referenced by other model elements in the same software model, the amount of reuse is calculated by counting the references to the model elements in other model elements. For example, in Figure 2-2a) the class model element named QueueElement is referenced in the method model element named Add as a local variable. This is an example of composition reuse by reference with different model element types (methods are not classes). In Figure 2-2b) the class model element named Queue is referenced by the class model element named PriorityQueue through inheritance. This is an example of composition reuse by reference with common model element types.

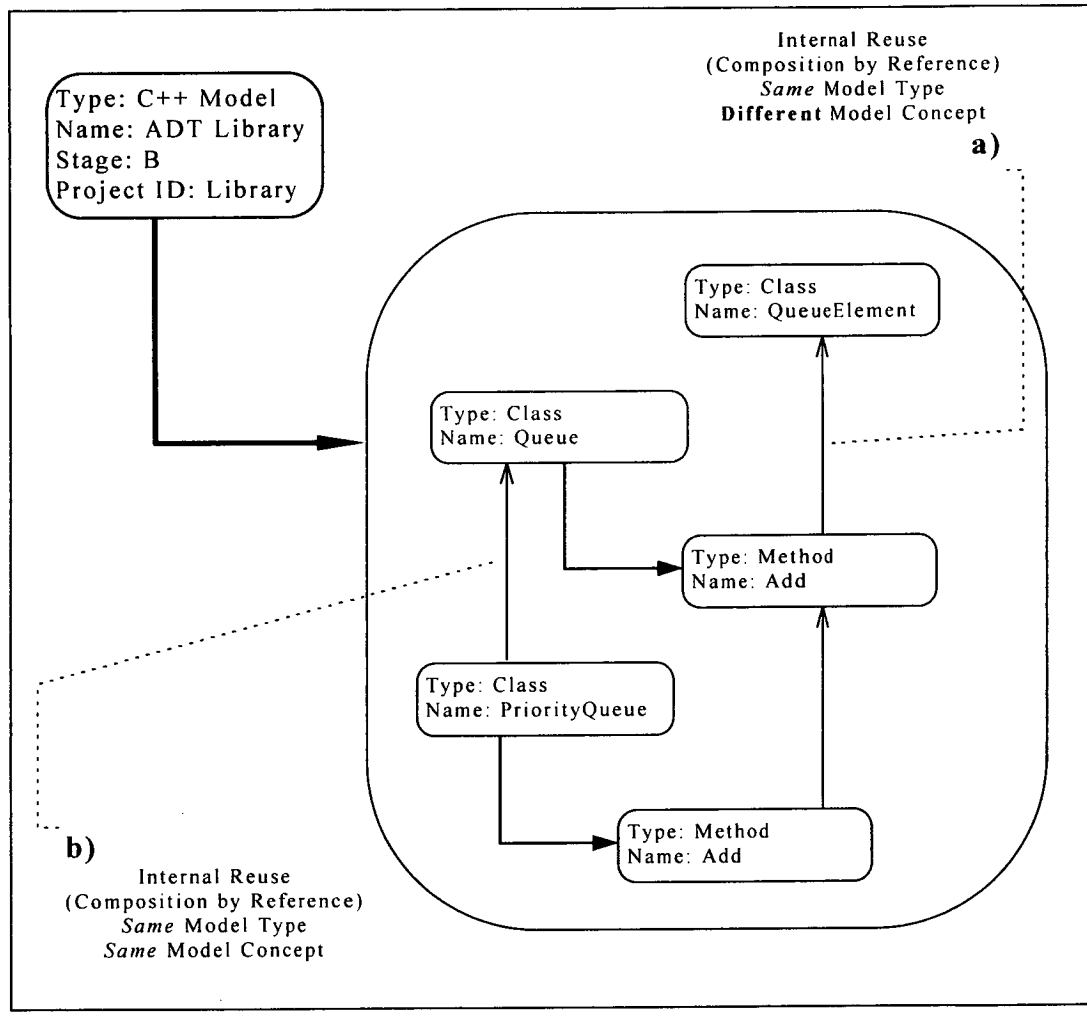

Figure 2-2: *Internal reuse based on reference*

Measurement of Traceability Reuse

External reuse using generation can lead to reuse via traceability (See Figure 2-3). Matsumoto [9] describes an example of this and a variation can be found in one other source [24]. The NATURE team are also working on this [15]. The library model that is reused in an application model can be traced to its complete model through internal reuse using generation or a traceability matrix for a previous project. All models indirectly linked to the source model in a project can be traced using the path of internal reuse and imported as draft software models in the current project.

This method has the benefit of constraining the search space for finding software models as well as reusing software models at many stages of development by just selecting one software model for reuse at one level. It is also the most complex to support through automation although it may support the development of application generators [13]. This form of reuse demonstrates one motivation behind requirements reuse. The amount of reuse is calculated by comparing the imported model with the final model in each instance of reuse. Composition reuse for each imported model can also be considered.

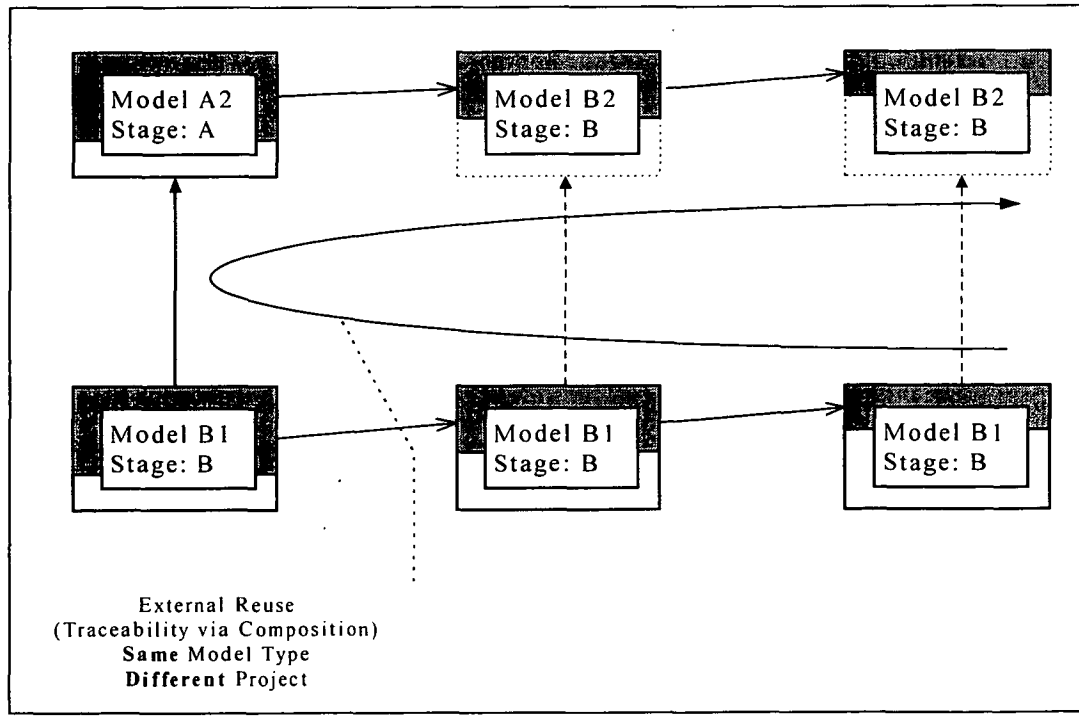

Figure 2-3: *External reuse using traceability*

Interpretation of Data

A measure of the amount of reuse needs some kind of interpretation for data analysis. In this thesis, the measure reflects the degree of labour intensity of software development based on reuse. For example, a high value of library code reuse in a program should reflect coding activity that is not very labour intensive, that is, little time spent in key tapping. A low value of library code reuse in a program should reflect coding activity that is very labour intensive, that is, much time spent in key tapping. In the case of reuse of designs or analysis requirements, key tapping can include certain mouse movements that are applicable to add, modify and delete menu items on a CASE tool.

Based on two sources [17, 25], three dimensions of measurement of the amount of reuse are suggested. These are:

- Assessment of Reuse Activity. The amount of reuse is one way in which the practice of software reuse can be verified. This includes development for reuse and development with reuse identified by Sommerville [26]. This helps determine if reuse is being practised in the development organisation.
- Assessment of Reuse Benefit. Given that reuse is being practised, the amount of reuse can be used as a correlative variable with various performance indicators such as productivity, defect rate, user reviews, and time to market. This is done to assess the benefits of reuse. It helps determine if reuse makes a difference to software development.
- Identification of Software Assets. Given that there have been some cost saving or benefit from reuse, the amount of reuse can help identify where such assets are located. It can assist in locating the software assets that improved performance.

2.3 The Basic Problem Defined

Before describing the problem of software reuse measurement, some more informal principles are introduced to describe its intuitive basis. Composition reuse can be viewed from the perspective of writing on paper. Assume that a book on software engineering management contains a chapter introducing some basic concepts in developing software. Assume then that another book on software reuse is written. The second book also contains the same chapter from the book on software engineering management. Thus, a chapter in one book is reused in another book. To assess the contribution arising from reuse (the amount of reuse), the number of pages for the chapter is divided by the total pages for the book, that is,

$$\text{Amount of Reuse} = \frac{\text{Number of Pages (Book Chapter)}}{\text{Number of Pages (Book)}} \times \frac{100}{1}$$

If the chapter is 20 pages long, and the book on software reuse is 300 pages long, the amount of reuse will be

$$\text{Amount of Reuse} = \frac{20}{300} \times \frac{100}{1}$$
$$= \frac{2000}{300}$$
$$= 6\ \%$$

The concept of generation reuse can be viewed from the perspective of automated manufacturing. Assume that a robot arm exists that can generate different kinds of nuts. Generation reuse occurs when the nuts generated by the arm contribute to a packet of nuts and bolts. The amount of reuse can be measured by comparing the volume of the nuts to the total volume of the packet of nuts and bolts, that is,

$$\text{Amount of Reuse} = \frac{\text{Volume (Nuts)}}{\text{Volume (Pkt of Nuts and Bolts)}} \times \frac{100}{1}$$

If each nut has a volume of 1 cubic centimetre, the volume of a packet of nuts and bolts is 130 cubic centimetres, and there are 20 nuts in a packet of nuts and bolts, the amount of reuse will be

$$\text{Amount of Reuse} = \frac{20}{130} \times \frac{100}{1}$$
$$= \frac{2000}{130}$$
$$= 15\ \%$$

Generation reuse can also be viewed from word processing. Assume that a letter wizard exists that is used to generate a draft letter, with the name, address, and start and end phrases. Assume that this wizard was used to generate a letter from a doctoral student to the head of school regarding a notice of intention to submit his thesis. The amount of reuse can be calculated by comparing the number of words in the generated draft letter with the number of words in a finished letter, that is,

$$\text{Amount of Reuse} = \frac{\text{Words(Draft Letter)}}{\text{Words(Final Letter)}} \times \frac{100}{1}$$

If the draft letter generated from using the wizard is 16 words long, and the letter sent to the head of school is 294 words long, then the amount of reuse will be

$$\text{Amount of Reuse} = \frac{16}{294} \times \frac{100}{1}$$
$$= \frac{1600}{294}$$
$$= 5\ \%$$

Please note that these are illustrative examples and are not intended to discuss the issues in measuring the amount of reuse for manufacturing or writing.

Now let us look at what happens when we try to make more consistent measures. Perhaps the measure is not accurate for generation reuse measurement of the bolts from the robot arm. If the nuts are made of steel and the bolts are made of titanium then the weight difference is not the same. A new kind of measure could be formed based on weight. That is,

$$\text{Amount of Reuse} = \frac{\text{Weight (Nuts)}}{\text{Weight (Pkt. of Nuts and Bolts)}} \times \frac{100}{1}$$

However, the volume difference cannot be denied. Thus, a second proposition could be made by combining weight and volume. That is,

$$\text{Amount of Reuse} = \frac{\text{Weight (Nuts)} \times \text{Volume (Nuts)}}{\text{Weight (Pkt. of Nuts and Bolts} \times \text{Volume (Pkt. of Nuts and Bolts)}} \times \frac{100}{1}$$

Again, even this measure can have variation. Multiplication is chosen in the above equation, but perhaps addition is better. That is,

$$\text{Amountof Reuse} = \frac{\text{Weight(Nuts)} + \text{Volume(Nuts)}}{\text{Weight(Pkt.of Nutsand Bolts} + \text{Volume(Pkt.of Nutsand Bolts)}} \times \frac{100}{1}$$

Similar considerations can be given to reuse measurement for books. For example, what if the book contains diagrams on some of the pages. This may lead to the need to count the words in the book, rather than the pages. That is,

$$\text{Amount of Reuse} = \frac{\text{Number of Words (Book Chapter)}}{\text{Number of Words (Book)}} \times \frac{100}{1}$$

However, diagrams can be complex and involve work, and they also need to be counted. Thus, a second equation can be arrived at for books with diagrams and words. That is,

$$\text{Amount of Reuse} = \frac{\text{Words (Book Chapter)} + (\text{Diagrams(Book Chapter)} \times 10)}{\text{Words (Book)} + (\text{Diagrams(Book)} \times 10)} \times \frac{100}{1}$$

The most difficult issue to deal with is changes to the final product that were contained in the original product. For example, what if the draft letter was generated in advance when the school was a department. The address of the letter needs to have a word changed from department to school. Using the calculation for reuse in letters, the amount of reuse would overestimated because the final letter still contains the same number of words as the original letter. What if the chapter in the reused software reuse book had half the diagrams changed but the number of diagrams remained the same, or words in parts of the chapter were changed?

What about different kinds of books? For example, a measure for words may be adequate for fiction books, but what about books of software modelling where diagrams are frequent but not included in the measure. Conversely, a book of house plans will contain few words and many diagrams making measurement of words superfluous. In addition, the reuse calculation for books containing diagrams does not consider the relative complexity of different diagrams. Compare the diagrams in Figure 2-1 and Figure 2-3 in this work.

The example of the different kind of book is very indicative of the measurement issue for reuse when dealing with different kinds of software models. We want to evaluate and compare different methodologies for writing different kinds of books. To do this we need consistent measures than can be applied to different kinds of books that measure reuse in a way that suggests a comparison using these measures is valid.

This problem arises in the literature for measurement of software reuse too frequently to consider the issue settled. Let us consider a real example. Let us say we used [27] to measure reuse based on structured methodologies with data flow diagrams and entity-relationship diagrams. We then change to a SAP based system and use measures form [28] to measure reuse. Banker counts the number of things reused verbatim. For example a data flow process is reused or it is not reused. If we reuse three of them and added three more then "reused object percent" = 50%. Daneva measures reuse at three levels and counts the size of the number of things reused in function points. Measures are provided to calculate reuse with minor modifications in addition to things that are reused verbatim. If we reuse two data flow processes verbatim that have a total size of 50 and reuse one data flow process with major modifications with a total size of 10 and add three more that have a total size of 100 then "Reuse (Verbatim)" = 50/160 (31%) and "Reuse (Major modifications)" = 10/160 (6%). Two problems occur here for comparing software methodologies. Firstly, Banker never differentiated between verbatim reuse measurement and reuse measurement with modifications. So we can't use the measures from Davena that consider modifications for comparison purposes. Secondly, even if we restrict the measures to verbatim reuse Banker does not consider the size of objects whereas Davena does consider them. This creates very different values percentage terms. What do we do when it comes to UML where measures for the amount of reuse, as at October 2003, do not exist? What about other methodologies that use software model types not contained in UML or covered by Banker or Davena?

Any software engineer will be found wanting; but not finding the same set of measures for a range of software model types. And yet we know that different kinds of software models have different kinds of things to measure even if the

measurement models strive to measure the amount of reuse. For example, data flow diagrams don't have classes and class diagrams don't have processes.

To have a framework that measures the amount of reuse using the same measurement model applicable to different kinds of software models, this is the problem this thesis has attempted to solve. Is this possible? Let us consider measurement of size and reuse for vehicles. We can measure the size of any kind of vehicle (car, truck) by weight or volume. We can identify the parts that any two kinds of vehicles have in common using the part numbers and measure the amount of reuse by volume or weight of these vehicle parts. This analogy suggests that it may be possible to do this in software engineering for different kinds of software models provided the size measure can be defined (weight, volume), and the means of identifying the common parts (part number).

2.4 Justification for a General Framework

Is there any evidence to justify a framework that measures the amount of reuse for different kinds of software models? This is the subject of section 2.4. If the reader is prepared to accept that measures for the amount of reuse must be defined for different kinds of software models then section 2.4 can be skipped entirely.

Short History of Software Methodologies

Over the past 50 years (1950 – 2000) we find that software methodologies were constantly improved to improve the outcomes on projects. Included in this variation of software methodologies is the variation in the kinds of software models used to specify and implement a software system. If the past is anything to go by, this would suggest that a general framework is needed to provide measures of reuse for different kinds of software models because they have constantly evolved over time. Let us now look at this proposition in detail.

Lowry *et al* [29] divide software methodologies into five groups. These groups are:

- Hierarchical Input Process Output (HIPO or HIPO-oriented)
- Structured process (process-oriented, functional development)
- Structured data (data-oriented)
- Object-oriented
- Post object-oriented (methodologies based on paradigms invented after object-oriented methodologies, that is, after 1986).

The first known software development methodology was developed by IBM and called HIPO. This methodology focused on what inputs were required to produce what outputs, and what processes were used to transform the inputs into the outputs. To main deficiencies with HIPO were a level of abstraction that did not capture a process and inadequate consideration of the structure of data. To address various deficiencies in the HIPO methodology, two other types of methodologies were developed. The structured process or functional development methodology was based on the work of Constantine and Yourdon [30] and DeMarco [31]. The structured data methodology was based on the work of Codd [32] and Chen [33].

The problems with structured process methodologies are that they do not address data abstraction or information hiding, they are not responsive to changes in the problem space, and they are inadequate for solving problems with natural concurrency [34]. Henderson-Sellers and Edwards [19] point out that the functional development methodology:

- has an emphasis on processes and neglects the data aspect of the problem,
- is based on modelling a system by a single function (which is not realistic, as normally a system is made up of many functions), and
- does not fully support or encourage reusability of components.

In contrast, the structured data methodologies, which deal effectively with data, do not fully address the processes associated with the data. The shift from analysis to design has also caused problems by not adequately defining the boundary or transition between analysis and design in structured process methodologies [35, 36].

To alleviate shortcomings in structured process and data methodologies, another set of software development methodologies, called object-oriented methodologies, has evolved. Abbot [37] may be the first to have developed an object-based methodology, although the notion of an object can be seen as far back as 1977, when it was used by Guttag [38] in defining abstract data types. Significant sources for object-oriented methodologies include Booch [39] and Rumbaugh *et al* [40] for analysis and design, and Meyer [41] for physical design and programming. McGregor and Skyes [2] is an early text that attempts to incorporate reuse into the methodology.

Object-oriented methodologies aim at modelling the problem domain through objects, which contain process definitions and data definitions as one unit, with the system viewed as a collection of these objects. Object-oriented methodologies are aimed at making design and implementation components reusable as well as making any system developed easier to adapt to changing demands. Object-oriented methodologies combine data and process modelling. Hence, they appear to overcome the existing mismatch between entity-relationship modelling of the structured data methods and data flow and functional analysis of structured process methods.

The object-oriented paradigm had its beginnings in programming languages like Simula 67. Simula 67 served as a foundation for SmallTalk and the popular C++ programming language, whereas C++ provided a foundation for the more recent Java programming language. Until the mid 1980s, much of the work in object technology was of an individual nature. The work of Guttag [38] and Abbot [37] typify early object research efforts.

The year 1986 marked the turning point with the launch of the ACM conference on Object-Oriented Programming: Languages, Systems, and Applications (OOPSLA).

This was the first conference dedicated to the object-oriented paradigm. In the same year, a landmark article by Grady Booch [34] may well be the first literature source to use the phrase "Object-Oriented Development" and discuss the use of the paradigm as an analysis and design method. A year later (1987) saw the launch of the European Conference on Object-Oriented Programming (ECOOP). Booch later published two widely cited texts on the subject [11, 39]. Although the titles may be misleading, both ECOOP and OOPSLA conferences contain material on object-oriented analysis and design.

With the launch of his 1988 book, Meyer [41] made the object-oriented paradigm a design as well as programming method and introduced the notion of design by contract. Shlaer and Mellor [42] did describe their analysis methodology as object-oriented, but one source did not agree that this was the case [35]. Coad and Yourdon [35] published what may be considered the first book that was object-oriented including the title ("Object-Oriented Analysis"). Another by book by James Rumbaugh *et al* [40] had considerable influence on the field with their Object Modelling Technique (OMT). Two other key texts appeared in this period, one by Rebecca Wirfs-Brock *et al* [43, 44] that introduced responsibility driven design, and one by Ivar Jacobson *et al* [45] that addressed requirements modelling with use cases. Coad and Yourdon [35, 46], along with De Champeaux *et al* [47], gained some significance in 1995 when their work was used by Humphrey [48] in the Personal Software Process.

In addition to the above works, the early 1990s saw the publication of many books on object-oriented software methodologies. The year 1994 saw the launch of the International Conference on Object-Oriented Information Systems (OOIS). This conference had an emphasis on information systems areas, such as object reuse, knowledge management, business process redesign, and technology transition. The early 1990s were marked by a plethora of object-oriented methodologies, each with their own notation, model, terminology, and process. In an effort to bring order to the field, some researchers developed taxonomies and made comparisons between different approaches to object-oriented development (For example, [19, 49-68]).

The maturing of terminology, models, and notation was eventually realised with the emergence of a standards organisation for distributed objects [69]. This standards body, the Object Management Group (OMG), was founded in 1989. By the end of 1997, this group had introduced standards for distributed objects [70, 71], object-oriented modelling [72], and CASE tool integration for object-oriented software models [72, 73]. The OMG is an open, international organisation with membership of individuals, universities, industry groups, and end users from all over the world. This organisation has contributed considerable stability to the diffusion of object

technology, particularly with reference models, terminology, and notation. Originally called the Unified Method, the Unified Modelling Language (UML) was launched by Grady Booch [39], James Rumbaugh [40], and Ivar Jacobson [45] in object technology at OOPSLA in 1995. Further on, more OMG members became involved and UML, now at version 1.1 [74-79], has become a major *de facto* standard for object-oriented modelling within OMG standards.

It is clear that software methodologies and the software model types used in them has changed over time. The impact on software reuse is that different software model types could have been reused. With this we derive the need for measures of reuse applicable to different software model types. Thus we can say that in the past a general framework was needed that could measure reuse for different kinds of software models as they evolve. i.e. a framework that can measure the amount of reuse for software model types in the future, not just software model types in the past or present.

The counter-argument is that the emergence of UML as a de facto standard implies a maturing of software modelling and that variation of software model types will decline. This implies that we only need reuse measures for UML because there will be no other software model types in use outside UML. Let us now consider both the variation in object-oriented methodologies and standards for object-oriented software model types to see how this affects the case for a general framework to measure the amount of reuse.

Variation in Object-Oriented and Software Methodologies

Upon examination of recent literature on software methodologies we find three significant findings:

1) There is much variation in object-oriented methodologies and many of these differences are real differences (i.e. non-superficial). If these variations are reused then measurement of the amount of reuse needs to account for these variations.

2) Software methodologies continue to be refined and this includes software model types that are completely new or significant refinements of previous software model types. If these variations are reused then measurement of the amount of reuse needs to account for these variations. Some authors even suggested that object-oriented modelling needed refinement.

3) UML does not contain a range of modelling constructs needed for the various kinds of software models in circulation. If these modelling constructs are used then measurement of reuse that is restricted to UML will be inadequate.

To illustrate this point, if measurement of the amount of reuse was restricted to classes then the measurement for reuse of statecharts would always be zero. This is because statecharts do not have classes, no matter how many states, transitions, events or actions in one statechart are reused in another statechart. Hence, measurement of the amount of reuse for statecharts needs to consider concepts such as states, events, transitions and actions. Let us now look at each of these findings in detail.

Variation in Object-Oriented Methodologies

Variation in software model types used in object-oriented methodologies is demonstrated by:

- Variation amongst more classic sources [2, 4, 11, 35, 39-42, 44-47, 80-82], both within and around the object technology community there are many variations to the modelling paradigm. This is demonstrated by:
- Numerous taxonomy and comparison papers that describe variation in more classic texts as well as other modelling approaches. Taxonomy literature sources can range from:
    - **a)** Notation comparisons [67].
    - **b)** Analysis [56, 68, 83].
    - **c)** Analysis and design [51, 55, 61, 66].
    - **d)** Terminology [63].
    - **e)** Design and programming [50, 65, 84].
    - **f)** Life cycle models [19].
    - **g)** Persistence [49, 53, 54, 58, 59, 62, 64, 85].
    - **h)** Concurrency [52, 57, 60].
    - **i)** Methodology [86] (one of the most comprehensive surveys).

Each of the variations provide some kind of extension to implementation or modelling. This leads to different software model types. For example, a class model can have *classes* with *attributes* and *operations*, and can include *inheritance*, *aggregation*, and *association* relationships. Aggregation can be further specialised with *Owns*, *ExclusiveHas*, and *Member* aggregation [87]. Use case diagrams can have *actors*, *use cases*, along with *uses*, *communication*, and *extends* relationships. Use cases can be further extended with *scripts* [88].

Variation in Software Methodologies

To find variations to software model types used in other software methodologies we have:

- A number of alternative or hybrid modelling methods. These include FDD [89], the living systems approach [90], systems oriented analysis and design [91], the CO-IP model [92], the L-G model [93], Integrated Modelling [94], and SBM [95].
- Refinements outside the object-oriented paradigm [15, 96-103].

Here are some brief descriptions of variations to modelling and implementation:

1) Bloom [104] describes a language for database programming and construction. Laenes [105] describes OOPS+, another object-oriented database programming language. Lee [106] describes OLI, a programming language that integrates logic and object-oriented programming.

2) [107] makes use of the MOOD methodology that extends object modelling with request-flow diagrams. [108] introduce the concept of a subject as a view of a class that has the same name but varying behaviour. For example, the class car for a car manufacturing system may be vastly different from the class car in a car rental system. This is similar to the concept of views in object-oriented databases [109, 110], where view represents a relative perspective of part of a set. [111] extends this with dynamic assertions. The impact of views culminates in a proposal of representation of viewpoints for views [112], placing the concept in the modelling extension category. [113] also extends the concept of subjects as views by showing how different interfaces need to be accounted for in design. [114] makes use of roles as a basis of identifying the subjective view of a class based on its relationship to other classes.

3) [115] introduces an object-oriented real-time programming language with its own syntax and semantics. [93] introduces a language graphic model designed specifically for analysis modelling of CASE tools. [116] describes a language-database integration model using "composition filters".

4) [117] shows how data flow analysis models can be transformed into object-oriented designs. [118] introduces a specification language for object-oriented analysis and design. This may be an alternative to OOSDL [80]. [119] describes DSM, yet another modelling language making use of Object Modelling Technique (OMT) and programming constructs. [120] makes use of software execution graphs as well as Data Flow Diagrams (DFDs) that have message flows for modelling of real-time embedded software.

All this further demonstrates that variations is software model types persist and that a general framework for measuring reuse with these software model types is desirable to make meaningful comparisons. Perhaps more significant is that some of these authors describe object models as inadequate for software development [90-92]. This suggests that software model types are still evolving as they have in the past and this also reinforces the case for a general framework that can measure the amount of reuse for different kinds of software models in the future.

The case for a general framework to measure reuse for different kinds of software models seems reasonable in light of the previous examination. However, the work done in the OMG, COMMA, and UML is based on the premise that there is considerable redundancy between various approaches to object-oriented methodologies [22, 69, 74, 86, 121, 122]. Are the differences just superficial or is there some deeper cause? Are the above results a manifestation of augmentation and extension to the paradigm [123]? Let us now consider UML in the light of the above findings.

Object Management Group and UML

The work done in establishing UML [74, 121] has genuine intentions: minimisation of redundancy on different approaches to modelling for object-oriented methodologies. If UML is stable and all-inclusive then this suggests that there is no need for a general framework that can measure reuse for a range of software model types. All we would need to do is develop a framework to measure reuse of UML. If this is the case then we should not be able to find any software model types absent from UML. What we find is that a number of extensions to modelling are still not included in UML version 1.1. Here are some significant examples:

- NATURE [15] still supports older modelling approaches with augmentation, these do not appear in UML as separate models.
- Graham's scripts not included [88].
- Concepts from viewpoints are not included [105, 110-112, 124]. This includes a recommendation for graphic representation of viewpoints [112].
- Software execution graphs and DFDs with message flows from Ellison [120] are not included.
- Business rules from Herbst [125] are not included. Herbst et al [126](p. 43 - 44) appear to favour modelling business rules explicitly or using methods outside the object-oriented paradigm, in this case BIER.
- Concepts from subject-oriented programming are not considered where the perspective of a class and its methods is dependent upon the perspective of the

user. For example, an application for transporting trees will consider the volume and weight of trees, an application for selling trees will consider the sale price and cost price. (See [108, 113, 114]).

- Request-Flow Diagrams are not included [107].
- Logic and Rule based paradigms (e.g. [106, 127]) are not accommodated.
- Although some petri-net concepts from [128] exist in UML [129], they do not exist as a modelling approach within UML. This includes extensions for distributed system specification and design [130, 131].
- Aggregation extensions are not included [87, 132].

Although some of the above examples are not "pure" object-oriented, reuse of models based on these paradigms still requires measures based on the varying structure of different software model types. If we only have measures for UML then we cannot detect reuse of software model types based on these variations. UML could be extended to include the above variations, but is this the way to deal with them? Measures for the amount of reuse would still be required for the variations. Is the reason of such variation in modelling due to the essential need to continuously refine and evolve different kinds software models to improve outcomes on new and different kinds of projects? This issue is dealt with under the heading "Tailoring in software methodologies", but before this is done, there is one more issue to address with regard to standards for object technology that could also provide a counter-argument to the case for a general framework to measure reuse.

In the OMG, standards for object interfaces are defined only for the design interface stage of development (detailed design) using IDL [70]. This is seen as the most practical way to promote reuse through portability and interoperability in addition to design reuse using composition. The OMG needs to consider mappings between different languages and mappings have been done for non-object-oriented languages. Hence, a certain level of unity can be achieved by using IDL for design interface reuse and we could restrict measurement of reuse to IDL and forget about reusing high level design or analysis specifications.

However, reuse of analysis and design models is increasingly being advocated. Examples include the NATURE team [15] and the REBOOT project [25], an early text on object-oriented methodologies [47], and one text that uses UML for software reuse [133]. The motivation and significance of reuse at earlier stages is based on the cost of defect removal. If the cost to fix a defect discovered during analysis is $1,000, then the cost to fix the same defect discovered during testing can be as much as $20,000 (pre-release), and up to $5,000,000 post release [134]. If we reuse analysis specifications that contain no defects then we are able to gain cost savings on defects

already removed. This is a benefit of software reuse that is evident in Japan [26, 135]. If we just used IDL then we would limit our cost savings to detailed design or code, that is, defects can still be injected in specifications that cost $20,000 to fix during testing if we just reused IDL and verified that this was done. If we reuse analysis specification then some of these costs are avoided.

We have observed that variation in software methodologies and software model types occurred in the past and persists in the present. Let us now consider a reason to suggest that variation in software model types will continue in the future. What is this reason? It is the need to tailor software methodologies.

Tailoring of Software Methodologies

The main aim of this part of the literature review is to identify the impact of tailoring of software methodologies and from this to identify its influence and implication on measurement of the amount of reuse. The important difference between this section (2.6) and the previous sections (2.4, 2.5) is that up to now variation of software model types is shown to be an observed phenomena in software methodologies in the past and present. Tailoring of software methodologies gives a reason for this phenomena and a reason to suggest that variation of software model types will continue to occur in the future.

The Basic Idea of Tailoring Software Methodologies gives a basic introduction to the concept of tailoring software methodologies.

A More Technical Introduction provides a more technical introduction to the concept of tailoring software methodologies with an emphasis on how tailoring causes variation in software model types.

Different Software Model Types and Implications for Reuse Measurement presents findings that indicate

- tailoring of software methodologies is the reason or cause of variation in software model types, and
- this is an essential feature of software methodologies that will persist in the future.

The implication for a framework that measures the amount of reuse is that it must be able to do so for a range of software model types.

The Basic Idea of Tailoring Software Methodologies.

Tailoring of software methodologies can be described as follows:

> *"Software methodologies are tailored when they are changed to better solve a given problem. This is manifest in better values for performance indicators."*

Essential performance indicators can be summarised as delivery of software on time and on budget that meets requirements and provides value to the user. Pfleeger [136] (p. 540) states that researchers and practitioners need to explore both the similarities and the differences in software engineering with other engineering disciplines. Wasserman [137] illustrates some of the similarities. Tailoring of software methodologies may well highlight one of these differences.

Other engineering disciplines may suffer slight shifts in their domains. For example, new ways to build bridges or computers. What does not happen in other disciplines is the use of civil engineering disciplines to make electronic circuits, or manufacturing methods for bridge building. In software engineering, it is possible to design a software system that either assists in building bridges or designing electronic circuits. Moreover, this can be done using different methods. This can also be considered from the concept of functionality. The limits of functionality for things such as bridges and cars are reasonably well defined. For example, bridges must allow users to pass the obstacle, cars must allow users to travel for greater distances using less energy. Software systems, on the other hand, can support a variety of varying functionalities. For example, software systems may need to support accounting functions, or control a manufacturing plant, or support decisions of executives. Until adequate knowledge is gained, tailoring of software methodologies appears to be one of way to gather this knowledge and cope with the current lack of it.

A key point worth noting is indicated in Wastell [138]. The effect of trusting a software methodology for all occasions cannot be denied. Belief in procedure rather than critical use and evaluation of a software methodology can lead to disastrous consequences, particularly if it approaches blind faith evident in data found by Wastell.

> "...One of the developers is quoted as saying: ...He said that we should all trust in the methodology and that it would all work out in the end... Of course, it didn't!..." (p. 30)

Sources on tailoring of software methodologies emphasise evaluation and modification, rather than rigidity and belief. [138] (p. 39) admits that he is somewhat anti-method, but this perspective is not entirely divorced from more structured approaches to software engineering. [48] states that,

> "...Software design is a creative process that cannot be reduced to a routine procedure. The design process, however, need not be totally unstructured..." (p. 310)

Perhaps what is important is not being always systematic or always creative, but to know when and where creativity and routine procedure are most effective in developing software systems that are on time, on budget, meet requirements, and are valued by the user.

A More Technical Introduction

A software methodology is tailored for some kind of reason. For example, the kind of application domain (finance, engineering) or the type of client organisation (large, small). To tailor a methodology we adjust some of its facets. For example, what kind of software model types are best suited to the project. If one wants to build software for a life support system then reliability of 100% is critical. But for a web-based personal banking application reliability of 80% or 90% is tolerable but throughput time is important to ensure clients use the software. For these two different kinds of applications we may look at different ways of specifying a system. We may use data flow models for the life support system because we know how to specify systems that are 100% reliable using data flow models but not object-oriented models. We may use object-oriented models for the web-based banking application because we know how to specify systems with acceptable post-release defect rates and better throughput times compared to data flow models.

An early description of the need to tailor software methodologies can be found is Senn [139]. In reviewing different methods for information systems development, Senn appears to support a number of aspects related to tailoring software methodologies. These are:

- Suitability of methods is based on the kind of application.
- Variation occurs in practice.
- Methods may need to be modified for better suitability.
- Results of applying a method determine its suitability.

This last inference points to evaluation in practice.

Consider the following quotes from Senn.

> "...There is no one right way to develop an information system, although there are ways to produce the right system for an application. Many variations on the development methods discussed in this chapter occur throughout the business community. Some methods are more successful than others, depending on when they are used, how they are applied, and who is involved in the development process..." (p. 40 - 41) (Suitability and Variation)

> "...In some instances, a step-by-step approach, comparable to the systems development life cycle, will be the only appropriate way to proceed. In other cases, prototyping will be the only method that makes sense. In still other situations, these methods will be combined and, in addition, users may develop part of the application themselves, perhaps using spreadsheets and a personal computer..." (p. 41) (Modification)

> "...The ultimate determinant of success for a particular development method is the results [sic] obtained, not the theoretical 'correctness' of the method..." (p. 41) (Evaluation)

A good example of tailoring software methodologies in practice can be found in [120]. This work also highlights the role and limitations of UML as a single standard modelling notation. Ellison's [120] methodology is made to handle the problems facing software developers who make software for real-time embedded systems for computer hardware, including driver card software for personal computers. [120] describes the methodology as "...tailored to the market-driven company..." (p. 2). Moreover, Ellison uses a unique variety of software model types with object-oriented and structured process models as well as software execution graphs. This work typifies what happens to software model types in software methodologies when they are tailored to a domain, organisation, or project. From this source we have:

- Extension of software model types: DFDs are extended with message connections and posted events between processes [120] (p. 140 - 141).
- Restriction of software model types. Ellison [120] (p. 195) recommends that DFDs only go to two levels for requirements analysis.
- Different combinations of existing software model types. DFD processes are components and each one is related to a number of classes using Rumbaugh et al [40] for notation [120] (p. 46 - 47, 56).
- Introduction of new software model types. Software execution graphs along with heavy use of tables for events and responses are introduced [120] (p. 23 - 24, 67). These are critical for performance evaluation of software prior to production of source code.

- Exclusion of software model types. There are no Use Case models in Ellison [120].
- The process varies considerably. The unique combination of modelling approaches has made a process different from other texts on object-oriented modelling [120] (p. 7 - 14).

This variation of software model types in Ellison [120] is reminiscent of the variations in software methodologies described earlier. Given that tailoring of software methodologies includes the adjustment of the software model types to enhance its performance relative to its application, this may be a major cause of the variation in software methodologies, including object-oriented software methodologies.

Let us now briefly consider the literature that supports the need to tailor on software methodologies and with this the need to modify and extend the software model types used in software methodologies.

Different Software Model Types and Implications for Reuse Measurement

So what sources support the need to tailor software methodologies? Can we find can we find sources on software measurement? can we find sources from software process improvement? can we find sources devoted to tailoring of software methodologies, can we find empirical studies that support the need to tailor software methodologies? can we find sources that automate tailoring of software methodologies, can we find sources on object-oriented methodologies, can we find other sources on software engineering, and (most importantly) and can we find sources that support the need to modify the software model types as part of tailoring software methodologies?

Well, we found sources in all of the above areas. Let us first cite a selection of sources that are a cross section of the above areas that support the need to tailor software methodologies.

- Software measurement [140-148]
- Software process improvement [48, 149-151]
- Sources devoted to tailoring of software methodologies [152-161].
- Empirical studies that support the need to tailor software methodologies [162-164]
- Automation of tailoring of software methodologies. [140, 165-174]
- Object-oriented methodologies [2, 74, 79, 88, 175-191]

- Other sources on software engineering
    - Analysis and design [139, 192]
    - CASE technology [10]
    - Software engineering [26, 134, 136, 137]
    - Requirements engineering [15, 193]
    - Software reuse [25, 133, 135].

Now let is cite a range of sources that support the need to either:

1. Select appropriate software model types,
2. Modify or extend software model types, or
3. Create new software model types

as part of tailoring software methodologies

- Software measurement [140, 141, 143, 146]
- Software process improvement [149, 150].
- Sources devoted to tailoring of software methodologies [152-157, 161].
- Empirical studies that support the need to tailor software methodologies [162, 163].
- Automation of tailoring of software methodologies [140, 165, 166, 169-174].
- Object-oriented methodologies [2, 76, 88, 92, 107, 122, 175, 177, 181, 182, 184, 187, 189-191, 194-200].
- Other sources on software engineering
    - Analysis and design [192]
    - CASE technology [10]
    - Software engineering [134, 136, 137]
    - Requirements engineering [15, 193]
    - Software reuse [133, 135].

There is considerable evidence to suggest that tailoring of software methodologies is necessary for their successful application as measured by meeting requirements on time and on budget. Tailoring of software methodologies has become so prevalent that a research area devoted to it has appeared using the term "method engineering" in [155, 156]. Note that six of the cited sources were published before 1990 (These are [142], [160], [149], [139], [10], and [141]). This suggests that the need to tailor software methodologies may have been the cause of variation in software

methodologies and software model types in the past. Given that variation in software methodologies and software model types persist in recent literature for the same reason (tailoring), this does add weight to the view that software methodologies, including software model types, will continue to evolve and vary in the future.

Let us now briefly cite and quote a few significant sources to add some substance to the proposition that

1. Software methodologies need to be tailored to be more suitable to different kinds of projects.
2. Software model types are selected, modified, or extended as a consequence of the need to tailor software methodologies.
3. Any framework for measurement of the amount of reuse needs to handle different software model types; past, present, and future.

The sources that support the above proposition are

- **An Action Research Study** by Akist and Bergmans [177],
- **UML Standards** and the Object Constraint Language (OCL),
- **The IFIP Report** in a framework of information systems concepts,
- **The Capability Maturity Model** [149], a widely accepted standard for improving software quality, and its relationship with Methods Engineering, and
- **Software Reuse in Japan** [135].

An Action Research Study: In an action research study, Akist and Bergmans [177] applied a number of object-oriented methodologies to a number of system types (twelve in total). These system types or small groups of them could represent problem domains. They appear to have used tailoring to find the best solution for development of each system. Consider the following quote.

> "...Our intention was to combine what we considered to be the best of these methods. For example, we used Coad and Yourdon's layered approach and their hints for object identification [4, 5], adopted Booch's notation [2], employed the rules of Johnson and Foote [6], applied the Law of Demeter [7], incorporated the associations and the dynamic model of OMT [9], and included the collaboration graphs of the Responsibility Driven approach [10]... In cases where we could not find a solution to our problem, we referred to other related research..." [177] (p. 350)

Two significant findings from [177] are that:

1. Depending on the type of system, different problems were encountered.

2. In some of these cases, no object-oriented methodology or related literature appeared to solve them.

More importantly, results may well have been worse if some form of tailoring was not used. Formation of a standard for object-oriented methodologies with insufficient data and disregard for tailoring may result in disastrous repercussions [184], [183] (p. 56), [189] (p. 227 – 228), [190] (p. 203 - 204), [201] (p. 8), [202] (p. 44 – 45).

UML Standards: UML version 1.1 [74, 79] allowed for extensions to the software model types under the heading of general extension mechanisms. The general extension mechanisms are constraints, comments, element properties, and stereotypes. The UML summary document [76] is explicit about tailoring of the process to organisations and problem domains. The summary document also describes some mechanisms and terms for user-defined extensions to UML [76] (p. 7). These are:

- UML Variant (a language extension to UML), and
- UML extension (a set of "Stereotypes", "Constraints", and "TaggedValues" that extend UML).

In addition,

- two other minor extensions to UML were published [77, 78],
- UML was extended with the introduction of the Object Constraint Language (OCL) [75], and
- At the first international workshop on UML (UML '98), Ou [187] recommended extension to UML in the form of constraints (two new model elements) when using UML for object-oriented database design.

OCL did not appear in UML 1.0 (See [121]). OCL is a non-trivial extension and this is demonstrated by the addition of a separate publication on OCL by Addison-Wesley [203]. This positive response to the need to refine UML with OCL already provides evidence to suggest that software model types will continue to change in the future.

The IFIP Report: The IFIP supported the release of a report on "A Framework of Information System Concepts" (The FRISCO report, [157]). Two significant features of this report are the recognition of diversity in information systems development, and support for integration of different kinds of software models manifest in the two following quotations.

> "...It is well known that there exists a large variety of such conceptual bases (frameworks of modelling concepts, modelling approaches, meta-models) for modelling information systems... Although we claim that the existing diversity is too large, we do not deny that a certain degree of diversity is appropriate to cater for the different, special requirements of special sorts of information systems, such as plain database systems, time-oriented or planning systems, document retrieval systems, rule-based systems, etc. Consequently, we do not advocate having one and only one conceptual basis for modelling all the various sorts of information systems..." (p. 11)
>
> "...One thrust of the report is to provide an ordering and transformation framework allowing to relate the many different information system modelling approaches (i.e. sets of concepts for modelling information systems, meta-models) to each other..." (p. 1)

The Capability Maturity Model: Humphrey [149] introduced the Capability Maturity Model (CMM) with five levels as part of managing the software engineering process. This text is not explicit about tailoring of the process or product to application or problem domains. However, there are two aspects of the text worth noting:

1. At level three, Humphrey introduces process definition. This is where a process for developing software is defined similar to a definition of a software methodology in method engineering. The process definitions include activities for the process and the products used and generated. Humphrey indicates that software engineering processes may need to vary according to the software project (p. 247).
2. At level five, Humphrey describes the process as an optimising process that constantly improves on previous processes for software development. Work in method engineering aims at optimising a software methodology to a problem domain or project.

In a later text Humphrey in the Personal Software Process to support CMM [48] (Table 13.1, p. 442) includes terminology for process definition. One of the terms is tailoring. This is described as follows.

> "...Tailoring... The act of adapting process designs and process definitions to support the enactment of a process for a particular purpose..." (p. 444).

Three other sources also assert links between methods engineering and CMM [150, 175, 176]. Rolland et al (Figure 1), appear to link the activities of a method engineer to levels in the CMM. Using a very similar diagram to Rolland et al, Odell [175, 176] (Figure 4) goes further and links levels four and five in CMM to method engineering.

Software Reuse in Japan: In a major text on software methodology practice in Japan Cusumano [135] (p. 9 - 12) surveyed a number of divisions within four major Japanese computer manufacturers (Hitachi, Toshiba, NEC, and Fujitsu) and included two common issues and features in their approaches to software methodologies. Firstly, software methodologies need to be tailored to different projects. To deal with this, the organisations provide some central R & D source to supply a resource of methods, tools, processes, and techniques to personnel in project development. These can, and usually are, tailored to the project. The approach is similar to the COMMA approach to standards for object-oriented methodologies [178], a core model that can be extended.

Secondly, each organisation has to deal with the impact of different kinds of systems or product types on reuse. Supporting reuse across different applications is in some conflict with their differences. One of the organisations (Toshiba) [135] (p. 258 - 260) appears to practise domain reuse described as white-box designs. Cusumano indicates that Toshiba may be a world leader in software reuse with the following quote:

> "...More focus in applications and hardware, in addition to a remarkably integrated set of tools, techniques, management procedures, controls, incentives, and training programs, facilitated Toshiba's efforts to systematize reusability of designs, code, specifications, and other elements, which it did probably as much or more than any other software facility in the world. Not all its product departments utilized available process technology to the same degree, reflecting the need to tailor practices to different application domains..." [135] (p. 218)

The last sentence in the quote is included to demonstrate that tailoring is still necessary even in a highly successful reuse-oriented software development organisation. Further examination of the source reveals the need to tailor software methodologies, including their products, to different domains. In 1986, Toshiba were delivering systems written mostly in FORTRAN (60%), but also in other problem oriented languages (20%) (p. 237 - 238). When describing automation support, Cusumano includes three features at Toshiba:

1. Product departments could add tools or methods that better suited their application domain (p. 249 - 250).

2. Automation supports various approaches to modelling requirements such as functional, contextual and dynamic (p. 235).
3. The use of "paradigms", which are models of software development tailored to a particular application (p. 258).

Evidence not only observation of variation in software model types but also a reason or cause for this observation appears to be tailoring of software methodologies. This leads to the following conclusion for measurement of the amount of reuse:

- Any framework for measurement of the amount of reuse in software methodologies should be able to measure the amount of reuse for different kinds of software models.

Does a framework exist that can measure the amount of reuse for different kinds of software models? this is the subject of section 2.5.

2.5 Measurement of Reuse for Different Software Model Types

In order to identify a framework for measurement of the amount of reuse with different software model types it was necessary to find some way to classify the literature. This was done because research in measurement of reuse had little evidence of cross citation and knowledge building. Section 2.7 is divided into three main sections:

A Model for Research Surveying: Presents a tool for formation of taxonomies to classify literature. This is used to form the taxonomy in section 2.7.2. This was done to provide more rigour to classification of the literature with the hope that the tool can be used by other researchers in other areas. The model for research surveying is used in the same way a software model type is used. We specify a research survey instrument for a subject using the model for research surveying analogous to specifying a model for a software system using UML.

Research Surveying Tool for the Amount of Reuse: Presents the taxonomy used to classify literature on measurement of the amount of reuse. The taxonomy was formed specifically identify any framework that can measure the amount of reuse for different software model types.

Literature Survey on Measurement of The Amount of Reuse: Presents findings that indicate no framework exists to measure the amount of reuse nor is there any measurement theory or model that has been sufficiently tested across a range of different software model types to ensure consistent measurement of the amount of reuse.

A Model for Research Surveying

One side effect of the proliferation of object-oriented methodologies in 1990-1995 was the publication for various taxonomies to classify them based on a set of concepts. For example, how many methodologies had the concept of a class or an active class? Further examination of these taxonomies indicates a common set of features used to form these taxonomies. For example, [68] used the feature of dimension and defined two dimensions to classify object-oriented methodologies, namely aspect and viewpoint. Some of these features can also be found in taxonomies that classify other areas of software methodology research.

When reviewing the literature on reuse measurement the work had a similar character to that of object-oriented methodologies in 1990 – 1995. Specifically, a range of material with varying terminology and little evidence of cross-citation or knowledge building.

We therefore decided to go one step further and define a model for research surveying that can be used to define taxonomies before defining a taxonomy to classify literature on reuse measurement. The model for research surveying is used to specify taxonomies for subjects in the same way UML is used to specify software models for systems. This was done for two reasons:

1. To provide more rigour for classification of sources
2. To provide a useful tool to the research community for classification of literature.

So what features can be found in taxonomies for classifying research in software methodologies? Table 2-1 summarises these features. Note well, it is the features themselves that are important, not the examples of the features. For example, "dimension" as a feature, not the example of a dimension, such as "aspect dimension" or "viewpoint dimension" in [68].

Table 2-1: Feature Identification based on Sources

Source	Location	Features
[204]	1-3, 5-7	Dimension, Subdivision, Vocabulary, Concept/Representation
[168]	76 - 80	Dimension, Subdivision
[66]	19 - 37	Subdivision, Vocabulary, Concept/Representation, Dependency
[93]	162-168	Dimension, Subdivision, Concept/Representation, Dependency
[205]	38	Aggregate dependency
[206]	28 - 38	Subdivision, Vocabulary, Aggregate dependency, Classification dependency
[19]	143 - 145	Aggregate dependency, Subdivision
[68]	156 - 158	Dimension, Subdivision, Vocabulary, Prerequisite dependency
[207]	106 - 109	Subdivision, Classification dependency, Vocabulary
[72]	5 - 7, 14 - 16	Subdivision, Vocabulary
[29]	223 - 224	Dimension, Subdivision
[208]	364 - 368	Classification dependency
[55]	36 - 40	Subdivision, Prerequisite/Classification dependency, Concept/Representation
[63]	38 - 42	Concept/ Representation

Location refers to page numbers. In some cases the feature is discussed (e.g. [204] for concept/representation), in others it is only evident (e.g. [55] for dependency)

Dimensions divide a taxonomy into mutually exclusive areas. [29] have methodology, organisation and linkage dimensions. [68], based on [56], has aspect and viewpoint dimensions. These are usually illustrated as lines at right angles with words (e.g. Figure 2-4; *Stages, Components*)

Subdivisions divide a dimension into smaller parts. [204] has a model level dimension with analysis and design subdivisions. [68] has structure, function, and behaviour subdivisions for the aspect dimension, along with individual and object community subdivisions for the viewpoint dimension. These are usually illustrated as intervals along the lines representing dimensions (e.g. Figure 2-4; *Analysis, Design, Modelling, Method Process*).

Vocabulary form the basic units for subdivision intersections. [68] has vocabulary for Individual structure (Object, attribute) and Object community structure (Inheritance, subsystem). [55] have a process subdivision with identify, placement and specification of classes. These are usually represented as words enclosed by areas delineated by subdivisions (e.g. Figure 2-4; *Class, Object, Operation, Aggregation, Identify Class, Identify Object, Identify Aggregation*). [207] enumerate the notion of vocabulary description. A conceptual vocabulary item has a "Name:" as the conceptual term, a "type" analogous to its subdivision, source contributions identified as "References", and information defining the term in "Description", "Notation:", and "Example".

A taxonomy can have a division between concepts and their representation. [63] indicates division between concept meaning (Object) and its varying terms (Instance, class instance, object). [66] has a conceptual model of the method process, and a representation model of modelling notation. [55] associate modelling concepts with their graphic representation.

Dependencies can further organise parts of taxonomies. Dependency types include:

Classification; one taxonomy part is a more general form of another taxonomy part. [208] has classification dependency for specific concepts, such as style (AI, Petri net, functional, OO) and notation (text or graphic).

Aggregation; one taxonomy part is composed of, or part of, another taxonomy part. [19] have subdivision aggregate dependency, where a phase (Build) is composed of more specific phases (Coding, Program testing, Program use).

Prerequisite or Existence; one taxonomy part is dependent on another taxonomy part for its own definition. [55] have the process and representation subdivisions dependent on object-oriented modelling concepts. [66] has the notation and method process for object-oriented methods dependent on their modelling concepts.

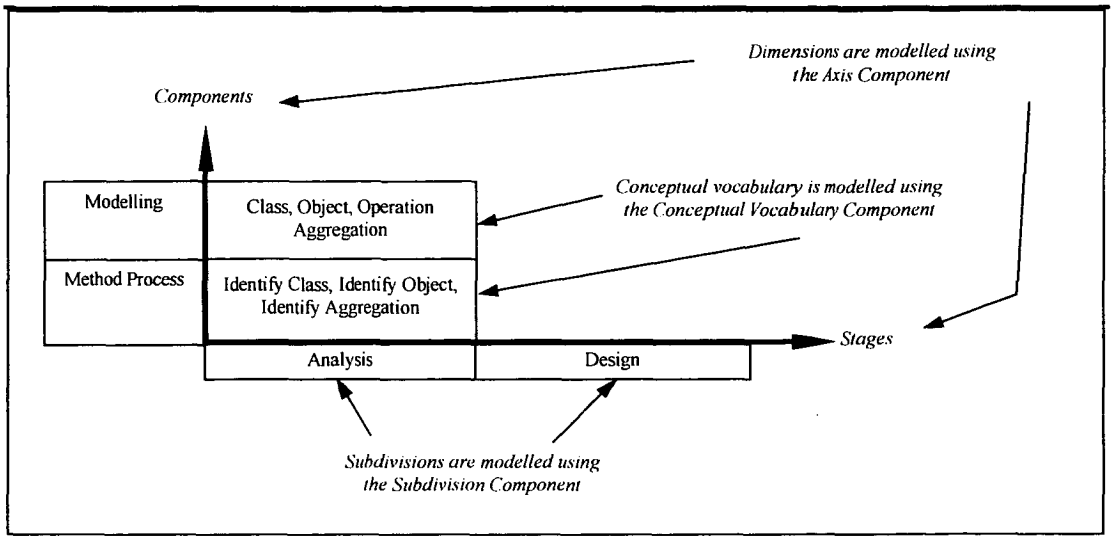

Figure 2-4: *Illustration of research surveying tool, analogous to a software application*

The above features appear across many taxonomies, however, each taxonomy usually contains only a subset of the features described, and lacks generalisation for research surveying in other subjects. How can these features be used to define taxonomies? In the same way UML defines notation for concepts used in object-oriented models, the model components define the notation for features used in taxonomies. Instances of these model components are defined by the researcher based on their knowledge of a subject and relevant literature. Whereas the model components define the structure of all taxonomies, instances of model components represent a particular taxonomy in a given subject. The relationship between the features and the model components is as follows:

- Dimensions are classified using the axis component.
- Subdivisions are classified using the subdivision component.
- Subdivision intersections are classified using the conceptual vocabulary component.
- Concepts, terms, and dependencies are classified within each axis, subdivision and conceptual vocabulary component.

An example of a model component and an instance of one is illustrated in Figure 2-5. Further details can be found in [209]. In this thesis, instances of the model components for classifying research into measurement of the amount of reuse are in Appendix B.

Conceptual Vocabulary Component

<Conceptual vocabulary area>
<Vocabulary dependency>
Vocabulary <Generic term>
<Description entry>

Term used	**Source**	**Evidence**
<Term used>	{ <Literature source> ; }	{ <Evidence statement> ; }

Conceptual Vocabulary Component Example

Analysis Composition Theory **Conceptual Vocabulary.**
Vocabulary Generic Term: Size
[**Description:** Size refers to the size of a software artefact for the purpose of measuring the amount of reuse.]

Term used	**Source**	**Evidence**
length	[Fenton, 1991 #499]	"Measures of attributes like... reuse are clearly important additional information to overall size... However, these should be considered separately...Specification X (or design Y, or program Z) has the following 'size: length is l units, functionality is f units... we might also wish to add that 'the amount of reuse is r1 units of total length and r2 units of total functionality..." (p. 156)
Number of Objects	[Banker, 1994 #508]	"...SRA computes a number of... reuse metrics that are based upon counts of new objects and reused objects in an application system..." (p. 179) "...INTERNAL REUSE PCT = 100% - NEW OBJECT PCT - EXTERNAL REUSE PCT... Internal Reuse Percent, here, is interpreted as the proportion of occurrences of objects written for an application (not counting the first occurrence of each object), compared to the total number of objects used in the application..." (p. 180)
Reusable Software Object	[Margono, 1992 #659];	"...The SPC and GTE-Contel models do not explicitly suggest a unit of measure, although we use the traditional SLOC as our unit of measure. The JIAWG model, on the other hand, explicitly defines a unit of measure called a reusable software object (RSO) which is a life-cycle product (requirements, designs, algorithms, code, test cases etc.) developed to be reused. It is really up to the user of this model to define what a life-cycle product is..." (p. 341);

Figure 2-5: Example of a Model Component and one of its Instances

Research Surveying Tool for the Amount of Reuse

The research surveying tool has two aims. The first aim is to identify and classify sources sufficiently to survey approaches for measurement of the amount of reuse (theory building). The second aim is to verify the feasibility of any given approach applied to different kinds of software models (theory testing).

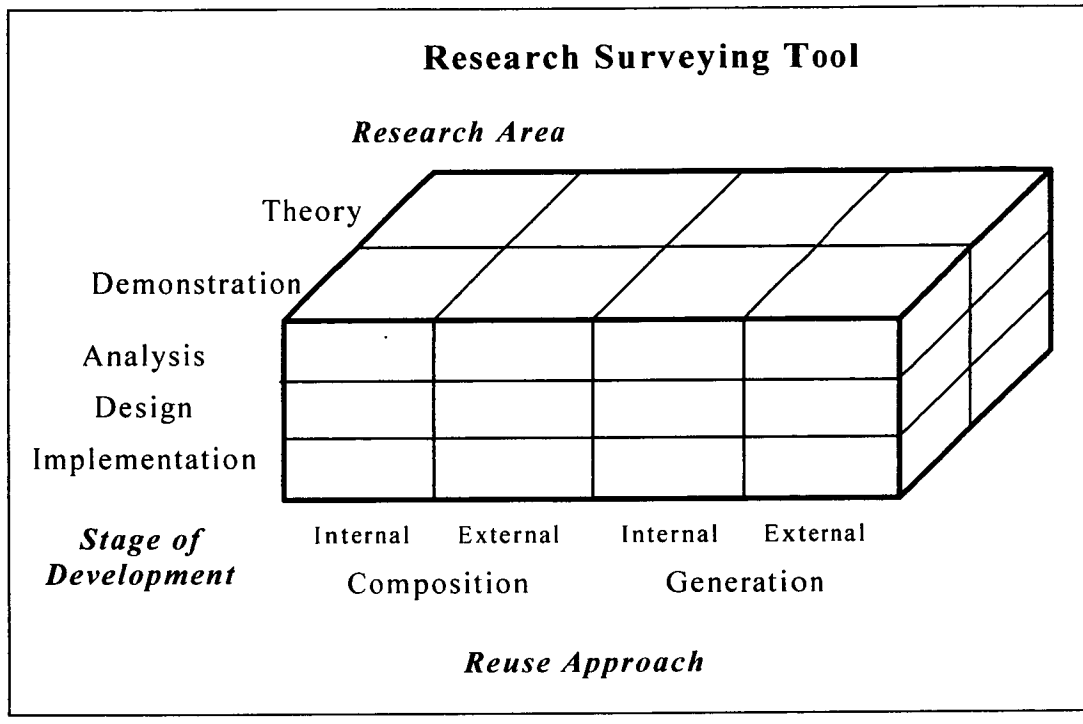

Figure 2-6: *Research Surveying Tool for the Amount of Reuse.*

The research surveying tool has three dimensions. The first dimension is the reuse approach. This dimension has four subdivisions. The first two are generation and composition. These two subdivisions in the reuse approach can also be subdivided by the other two subdivisions. These are internal and external. The second dimension is the stage. This refers to the stage of development a given product is created or reused in. The stage dimension has three subdivisions. These are analysis, design, and implementation. The third dimension is the research area dimension. This refers to the kind of research done. This dimension has two subdivisions. These are theory and demonstration. The theory subdivision classifies sources that are a theoretical explanation for measurement of the amount of reuse. For example, the amount reused could be calculated using Lines of Code Reused (LOCR) divided by Line of Code in the Program (LOCP). This would be a theory. The demonstration subdivision classifies sources that demonstrate a theory using some software model and software model type. For example, LOCR divided by LOCP could be demonstrated using the C++ programming language as a measure of the amount of reuse.

Sources classified using the theory subdivision identify the theories available (theory building). Sources classified using the demonstration subdivision illustrate use of these theories for different kinds of software models (theory testing).

Subdivisions from each dimension are used to form conceptual vocabulary areas, each with a number of concepts. For composition reuse, these are:

- **Size.** Size refers to the size of a software artefact for the purpose of measuring the amount of reuse.
- **Amount Reused.** Amount Reused is a measure of the amount that one software artefact contributes to another software artefact through composition reuse.
- **Amount Added.** Amount Added is a measure of the amount added to a software artefact after reuse of another software artefact through composition reuse.
- **Amount Not Reused.** Amount Not Reused is a measure of the amount of a software artefact not reused in another software artefact through composition reuse.

Table 2-2 names the conceptual vocabulary areas for composition reuse, along with the vocabulary concepts for each conceptual vocabulary area.

Table 2-2: Conceptual Vocabulary Areas and Concepts for Composition Reuse.

Concepts ⇒ Areas ⇓	Size	Amount Reused	Amount Added	Amount Not Reused
Theory				
Analysis Composition	X	X	X	X
Analysis External Composition		X	X	X
Analysis Internal Composition		X	X	X
Design Composition	X	X	X	X
Design External Composition		X	X	X
Design Internal Composition		X	X	X
Implementation Composition	X	X	X	X
Implementation External Composition		X	X	X
Implementation Internal Composition		X	X	X
Demonstration				
Analysis Composition	X	X	X	X
Analysis External Composition		X	X	X
Analysis Internal Composition		X	X	X
Design Composition	X	X	X	X
Design External Composition		X	X	X
Design Internal Composition		X	X	X
Implementation Composition	X	X	X	X
Implementation External Composition		X	X	X
Implementation Internal Composition		X	X	X

For Areas grouped under " Theory", the Conceptual Vocabulary Area name can be derived by adding "Theory" to the Phrase in the Area column. For example, Design Composition (Area grouped under "Theory") => Conceptual Vocabulary Area = Design Composition Theory.
For Areas grouped under " Demonstration", the Conceptual Vocabulary Area name can be derived by adding "Demonstration" to the Phrase in the Area column. For example, Design Composition (Area grouped under "Demonstration") => Conceptual Vocabulary Area = Design Composition Demonstration.

Concepts used for conceptual vocabulary areas in generation reuse are:

- **Size.** Size refers to the size of a software artefact for the purpose of measuring the amount of reuse.
- **Amount Generated.** Amount Generated is a measure of the amount that one software artefact contributes to another software artefact through generation reuse.
- **Amount Not Generated.** Amount Not Generated is a measure of the amount added to a software artefact after reuse of another software artefact through generation reuse.
- **Waste Generated.** Waste Generated is a measure of the amount of a software artefact not reused in another software artefact through generation reuse.

Table 2-3 names the conceptual vocabulary areas for generation reuse, along with the vocabulary concepts for each conceptual vocabulary area.

Table 2-3: Conceptual Vocabulary Areas and Concepts for Generation Reuse.

Concepts ⇒ Areas ⇓	Size	Amount Generated	Amount Not Generated	Waste Generated
Theory				
Analysis Generation	X	X	X	X
Analysis External Generation		X	X	X
Analysis Internal Generation		X	X	X
Design Generation	X	X	X	X
Design External Generation		X	X	X
Design Internal Generation		X	X	X
Implementation Generation	X	X	X	X
Implementation External Generation		X	X	X
Implementation Internal Generation		X	X	X
Demonstration				
Analysis Generation	X	X	X	X
Analysis External Generation		X	X	X
Analysis Internal Generation		X	X	X
Design Generation	X	X	X	X
Design External Generation		X	X	X
Design Internal Generation		X	X	X
Implementation Generation	X	X	X	X
Implementation External Generation		X	X	X
Implementation Internal Generation		X	X	X

For Areas grouped under "Theory", the Conceptual Vocabulary Area name can be derived by adding "Theory" to the Phrase in the Area column. For example, Design Generation (Area grouped under "Theory") => Conceptual Vocabulary Area = Design Generation Theory.
For Areas grouped under " Demonstration", the Conceptual Vocabulary Area name can be derived by adding "Demonstration" to the Phrase in the Area column. For example, Design Generation (Area grouped under "Demonstration") => Conceptual Vocabulary Area = Design Generation Demonstration.

Literature Survey on Measurement of The Amount of Reuse

The main concern of the survey is to see if research into measurement for the amount of reuse addressed the following key requirement:

- Any framework for measurement of the amount of reuse must cope with different kinds of software models.

[210] bears a strong resemblance to the key requirement identified in this study with the following quote.

> "...An appropriate metric needs to be able to take into account the unique differences of both the structured and object-oriented methodologies..." (p. 17).

[210] was referring to complexity metrics, but the need to account for different kinds of software models is clearly evident. The need for this is also evident for measurement of size when [18] is considered.

> "... The state-of-the-art for size measurement is that... there is some consensus view on measuring length of programs but not [of] specifications or designs..." (p. 157)
>
> "...Defining... length measures for specification and design documents is, unfortunately, not so easy... such documents consist of a myriad of text, graphs, and special mathematical diagrams and symbols. The nature of these will depend on the particular style, method, or notation used..." (p. 159 - 160)

This view is further supported by [211] (p. 318). It appears that there may be a fundamental need to make measures applicable to different kinds of software models, with amount of reuse as no exception. This only reinforces the need to meet the first requirement for measurement of the amount of reuse.

What does research on measurement of the amount of reuse indicate? Well, *key findings* of the survey indicate:

- No single set of measurement models is adequately tested against enough software model types to suggest it can cope with different kinds of software models.
- Measures for the amount of reuse are inadequate for object-oriented and UML software models.
- Use of various measurement models does not help because this can only lead to inconsistent data, both in terms of how it is analysed and interpreted, and how it affects data in other measurement models (e.g. Cost, Productivity) as an input variable.

These findings clearly indicate that we need a general framework to measure the amount of reuse for consistent analysis of data. These findings are illustrated in Table 2-4 to Table 2-13.

Some *additional findings* also worth mentioning are:

- Measurement models for the amount of reuse are insufficient for analysis and design reuse, internal and external reuse, and generation reuse
- Proposed general approaches to measuring reuse are not adequate
- Function points cannot be used because of too many anomalies
- Confusion of Measurement for Modification and Reuse
- Pre-mature validation can have long term consequences

Each one of these findings must be considered because they either:

- The findings have some impact on measurement of the amount of reuse and therefore any general framework for measurement of the amount of reuse must address them.
- The findings are potential counter-arguments to a general framework for measurement of the amount of reuse.

The *additional findings* can be skipped and the reader can still understand the greater part of the argument. However, if the reader is an avid researcher in reuse measurement then the *additional findings* are essential reading. Let us now look at each of these findings in detail.

Key Findings

No single set of measurement models is adequately tested against enough software model types

Table 2-4 tabulates real differences between tested theories for size. Each of these is given an ID that is used in each column in Table 2-5. Each theory tested is then cross-referenced against a software model type identified in each row in Table 2-5. For example, the length approach to measuring size (ID:1) is demonstrated using data flow diagrams. Table 2-6 and Table 2-7 tabulate real differences between tested theories for amount reused. Each of these is given an ID that is used in each column in Table 2-8, Table 2-9, and Table 2-10. Each theory tested is then cross-referenced against a software model type identified in each row in Table 2-8, Table 2-9, and Table 2-10. For example, the public reuse (P) approach to measuring amount reused (ID:11) is demonstrated using data flow diagrams. Table 2-11 tabulates real differences between tested theories for amount added. Each of these is given an ID

that is used in each column in Table 2-12. Each theory tested is then cross-referenced against a software model type identified in each row in Table 2-12. For example, the new object percent approach to measuring amount added (ID:1) is demonstrated using rule sets. Table 2-13 tabulates real differences between tested theories for amount not reused. As the table illustrates there is only one demonstration of its measure with the programming language BLISS.

Table 2-4: Measurement Models for Size

ID	Name	Description	Sources
1	length	Size of a product using some unit of measure.	[18]
2	Number of Objects	A product can be broken down into objects that are counted.	[27]
3	Size (S)	A product can be counted by calculating the weight of each of the components in the product and summing these values. Size (S) = $\sum$ weight (e)	[212]
4	Size (S)	A product can be counted by counting the size of each of the components in the product and summing these values. Size (S) = $\sum$ Size (c)	[213] [214]
5	LOC	A product that is source code can be counted by counting the lines of code in it. LOC = Line of Code	[17] [215] [143] [48] [216] [217] [211]
6	EASL	The size of any source code product can be determined by counting the number of equivalent assembly code lines for it.	[135]
7	Function Points	The size of a system based on the complexity of each system function.	[28]
8	Media Size	The size of a system based on the number of bytes.	[216]
9	Basic Count	The size of the system by couting a simple unit.	[216] [218]

Table 2-5: Demonstration of Theory for Size Measures

Measurement Model ⇒ Software Model Type ⇓	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Data Flow Diagrams	X								
Algebraic Specifications	X								
Z Schemas	X								
Data Dictionary	X								
Entity-Relationship Diagrams	X						X		
State Transition Diagrams	X								
Business Processes		X							
Rule Sets		X							
3GL Modules		X							
Screen Definitions		X							
Files		X							
Data Views		X							
Data Elements		X							
Data Domains		X							
Reports		X							
Report Sections		X							
C			X						
C++				X	X				
Ada					X				
COBOL					X				
Fortran						X			
High Level Languages						X			
Event Driven Process Chains							X		
Media Files								X	X
Cgi					X				X
Java					X				X
javascript					X				X
BLISS					X				

Table 2-6: Measurement Models for Amount Reused (1 - 10)

ID	Name	Description	Sources
1	Reused Object Percent	The number of objects reused as a percentage of the total objects used for a product. New Object Percent = Number Of New Objects Built / Total Number Of Objects Used * 100	[27]
2	Number of Objects Reused	The number of components not made as part of the product. (Derived Calculation) Number of Reused Objects = Total Number of Objects Used – Number of New Objects Built.	[27]
3	External Reuse Percent	The percentage of a product that is made from components of other products. External Reuse Pct = Number Of Objects Owned By Other Systems / Total Number Of Objects Used * 100	[27]
4	Number of Objects Owned by Other Systems	The number of components in a product that are owned by other products. Contained in the following equation. External Reuse Pct = Number Of Objects Owned By Other Systems / Total Number Of Objects Used * 100	[27]
5	Internal Reuse Percent	The percentage of components in a product that are reused more than once and are not owned by other products expressed as a percentage. Internal Reuse Pct = 100 – New Object Pct – External Reuse Pct	[27]
6	Reuse Metric	A component of a product is either reused (1) or not reused (0)	[219]
7	Reuse Rate	The percentage of reuse for a given product ReuseRate(S) = Reuse(S) / Size(S)	[17] [135] [213] [214]
8	LOC reused	The lines of code reused in a given product	[48] [215] [217] [216]
9	Four Point Scale	Reuse of components in a product can be classified into four levels: 1. Complete reuse without revision (0% changes) 2. Reuse with slight revision (< 25% changes) 3. Reuse with major changes (≥ 25% changes) 4. Complete new development	[220] [221] [213] [214]
10	Reuse (S, p)	Reuse of a product from another product can be calculated by counting all the components referenced in the former that are contained in the later. Reuse (S, p) = $\sum$ weight(e) where e is contained in p. Reuse (S) = Reuse (S, p)	[212]

Table 2-7: Measurement Models for Amount Reused (11 - 27)

ID	Name	Description	Sources
11	Public Reuse (P)	Reuse equals the length of the new part of the product plus the length of the reused part of the product, then dividing this into the length of the reused part of the product. public reuse (P) = length (P_2) / (length(P_1) + length(P_2))	[18]
12	AppReusePerc (S)	The percentage of reuse in a product that reuses other products. AppReusePerc(S) = Reuse (S) / Size(S)	[212]
13	RepReusePerc (S, p)	The percentage of reuse in a product that is reused in other products. RepReusePerc(S, p) = Reuse (S, p) / Size(p)	[212]
14	Verbatim Reuse	Verbatim reuse is where a product is reused entirely in another product.	[215]
15	Leveraged Reuse	Leveraged reuse is where part of a product is reused in another product and modified	[215]
16	Reuse (S)	The percentage of a product that is reused in another product less any changes made to the new product. Reuse (S) = (1 – %Change) * Size(S)	[213] [214]
17	Total Reuse Level	For each component that contains components in a product. The number of components in a component that are reused.	[12]
18	Total Reuse Frequency	The number of references to components in a component that are reused.	[12]
19	External Reuse Level	The number of components in a component that are from an external repository.	[12]
20	External Reuse Frequency	The number of components referenced in a component that are from an external repository	[12]
21	Internal Reuse Level	The number of components in a component that are used more than once.	[12]
22	Internal Reuse Frequency	The number of components referenced in a component.	[12]
23	Reuse Percent and level	Reuse of components at a given level of modification expressed as a percantage.	[28] [211]
24	Basic Reuse Count	A simple count of number simple units reused.	[216] [218] [222]
25	Reuse Density	The number of references to reused components expressed as a percentage of total size for the application model.	[223] [224]
26	Reuse Size and Frequency	The total size of the system subtract any additions expressed as a percentage of the total size of the system.	[224]
27	Reuse Percent	The size of reused units expressed as a percentage of the total size of the system.	[224]

Table 2-8: Demonstration of Theory for Amount Reused Measures (1- 11)

Measurement Model ⇒ Software Model Type ⇓	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Data Flow Diagrams											X
Algebraic Specifications											X
Z Schemas											X
Data Dictionary											X
Entity-Relationship Diagrams											X
State Transition Diagrams											X
Business Processes	X	X	X	X	X						
Rule Sets	X	X	X	X	X						
3GL Modules	X	X	X	X	X						
Screen Definitions	X	X	X	X	X						
Files	X	X	X	X	X						
Data Views	X	X	X	X	X						
Data Elements	X	X	X	X	X						
Data Domains	X	X	X	X	X						
Reports	X	X	X	X	X						
Report Sections	X	X	X	X	X						
C	X						X	X		X	
C++							X	X	X		
Ada								X			
COBOL							X				
Fortran							X		X		
High Level Languages											
Class Diagram						X					
PL/I							X				
ASS							X				
Basic							X				
Lisp							X				
Prolog							X				
Pascal									X		
Event Driven Process Chains											
Java								X			
Java Script								X			
Cgi								X			
Media files											
BLISS								X			
RPG											

Table 2-9: Demonstration of Theory for Amount Reused Measures (12- 22)

Measurement Model ⇒ Software Model Type ⇓	12	13	14	15	16	17	18	19	20	21	22
Data Flow Diagrams											
Algebraic Specifications											
Z Schemas											
Data Dictionary											
Entity-Relationship Diagrams											
State Transition Diagrams											
Business Processes											
Rule Sets											
3GL Modules											
Screen Definitions											
Files											
Data Views											
Data Elements											
Data Domains											
Reports											
Report Sections											
C	X	X				X	X	X	X	X	X
C++					X						
Ada			X	X							
COBOL											
Fortran											
High Level Languages											
Class Diagram											
PL/I											
ASS											
Basic											
Lisp											
Prolog											
Pascal											
Event Driven Process Chains											
Java											
Java Script											
Cgi											
Media files											
BLISS											
RPG											

Table 2-10: Demonstration of Theory for Amount Reused Measures (23- 27)

Measurement Model ⇒ Software Model Type ⇓	23	24	25	26	27
Data Flow Diagrams					
Algebraic Specifications					
Z Schemas					
Data Dictionary					
Entity-Relationship Diagrams	X				
State Transition Diagrams					
Business Processes					
Rule Sets					
3GL Modules					
Screen Definitions					
Files					
Data Views					
Data Elements					
Data Domains					
Reports					
Report Sections					
C			X	X	X
C++					
Ada					
COBOL					
Fortran					
High Level Languages					
Class Diagram		X			
PL/I					
ASS					
Basic					
Lisp					
Prolog					
Pascal					
Event Driven Process Chains	X				
Java		X			
Java Script		X			
Cgi		X			
Media files		X			
BLISS					
RPG			X		

Table 2-11: Measurement Models for Amount Added

ID	Name	Description	Sources
1	New Object Percent	The percentage of new objects in a product.	[27]
2	Number of new objects built	The number of components in a product made without reuse.	[27]
3	GPR-0	The number of source lines of code made.	[135]
4	Leveraged Reuse	The number of components reused in a product with modifications.	[215]
5	% Change	The percentage of change in a product after it is reused.	[213] [214]
6	Added LOC	The lines of code added to the system.	[217]

Table 2-12: Demonstration of Theory for Amount Added Measures

Measurement Model ⇒ Software Model Type ⇓	1	2	3	4	5	6
Business Processes	X	X				
Rule Sets	X	X				
3GL Modules	X	X				
Screen Definitions	X	X				
Files	X	X				
Data Views	X	X				
Data Elements	X	X				
Data Domains	X	X				
Reports	X	X				
Report Sections	X	X				
Fortran			X			
High Level Languages			X			
Ada				X		
C++					X	
BLISS						X

Table 2-13: Demonstration of Theory for Amount Not Reused

Measurement Model ⇒ Software Model Type ⇓	1
BLISS	[217] counts the number of deleted LOC.

What these tables illustrate is that coverage of theory testing is sparse for real differences in measures for the amount of reuse. To put it another way, for any given theory referenced in Table 2-5, Table 2-8, Table 2-9, Table 2-10, and Table 2-12, no column with a given ID has an X in each corresponding row for every software model type listed. Table 2-8 (ID: 1 to 5) and Table 2-12 (ID: 1 to 2) appear to show some promise for theory testing based on a measurement model from the same source [27]. The measures were defined for a specific set of software model types automated on a specific tool. What about object-oriented models, or Z, or even UML? In addition, Banker et al count reuse based on a single entity without regard to its size. How do reused relationships compare with reused classes? This is important because measuring the amount of reuse is about how much is reused not just what is reused. Most significant is that Banker et al do not describe how their framework can be used to measure reuse for new kinds of software models.

What are the implications of this? The critical concern is an inability to make reasonable comparisons for activity to assess development with reuse and development for reuse. To illustrate the problem, having different metrics for the amount of reuse plotted against performance indicators, such as productivity, is like having different measures of engine power plotted against speed. Two studies looking at engine capacity versus speed cannot be compared if one study uses horse power to measure engine capacity and the other study uses litres. The authors actually measure engine capacity in a different way. This is similar to comparing which car has better performance by using miles per hour for one car and kilometres per hour for another car. Consider the following analogy. If an engineer wants to know what effect increasing the carburettor's barrel diameter size has on speed and fuel economy and does this using two different kinds measures of diameter for two different cars, how meaningful will it be to compare two reports for each car that use these custom measures for the barrel diameter size?

Measures for the amount of reuse are inadequate for object-oriented and UML software models.

Another point worth noting is that Table 2-5, Table 2-8, Table 2-9, Table 2-10, and Table 2-12 indicate that theory testing for measurement of the amount of reuse using object-oriented models is limited to classes, attributes, and operations based on the inheritance hierarchy and communication at the implementation stage. Three other discoveries are of particular concern for object-oriented modelling and UML.

- Only nine sources found were applicable to measurement for object-oriented software models for theory building or testing [27, 213-215, 219, 221, 222, 225, 226].

- The only theory testing that includes UML to any degree beyond these concepts is one attempt for state transition diagrams [18].
- The remaining sources focus on classes, attributes, and operations, with five dealing only at the implementation level [213-215, 221, 222, 226].
- As of September 2003, no measures for the amount of reuse were found specifically for UML.

Even if the measurement models could be adjusted for software model types applicable to analysis and design, advanced concepts for state transition diagrams, class diagrams, and interaction diagrams are not dealt with nor is the object constraint language. This is not appropriate for measurement based on software model types at analysis and design stages, including UML. There is much more that can and cannot be reused in these models.

Use of various measurement models does not help because this can only lead to inconsistent data

Use of different measures as a means of coping with different software model types does not help. This could easily lead to inconsistency of data and meaningless comparisons. For example, if data about reuse for one methodology was gathered using the measures from [27] and this is compared to data gathered about reuse for the SAP methodology using the measures from [28] no meaningful conclusions can be made about the relative effectiveness of reuse using the amount of reuse as an indicator. In addition, the conversion is not as trivial as a multiplier of 1.6 to convert miles to kilometres.

If a means for doing this cannot be obtained, then comparison between different approaches for software development can never be obtained. The effect of this is illustrated in [210] for complexity. [210] describes the problem of what could end up happening when comparing two development paradigms for their capacity to cope with change using complexity. Because the complexity measures are so different it is not necessarily clear that one method is better than the other when it comes to coping with a change in requirements.

To put it another way, three field studies for software reuse ([219], [213, 214], and [221]) make use of different measures for the amount of reuse. If these were used as reports for assessment and improvement of practice, none of them could be compared to each other. Consider also the measures used on source code in two other experiments [212, 215]. One has an array of measures for each component in a software model, the other has measures for the entire software model with similar consequences. If these were used as reports for assessment and improvement of practice, no meaningful comparisons could be made. Although variation of theory is

in some ways a necessary part of early research, a comparison that is consistent needs to be done to conduct accurate assessment and improvement of a software process.

The significance of this is not to be underestimated when measures for the amount of reuse are fed into cost models and reuse economics models [23, 227]. [23] illustrate use of the amount of reuse measures as inputs to cost models such as SPC and JIAWG. [227] cites four reuse economics models that use the amount reused, referred to as reused lines of code in product or RLOC. [23] also consider reuse measurement beyond code as important, that is, requirements and designs as well as code are classified as reusable software objects when considering these cost models. If measures vary then data for cost models could end up all over the place, making results look ambiguous or meaningless for comparison. In addition, this can have dire consequences for estimation of cost that results in under resourced or over resourced of projects.

Additional Findings

Measurement models for the amount of reuse are insufficient for analysis and design reuse, internal and external reuse, and generation reuse

The number of sources that define of some model for measuring analysis or design reuse, internal or external reuse, or generation reuse is very small, Table 2-4 and Table 2-5 illustrate this. Table 2-4 lists all citations by reuse approach. For example there were 29 citations that describe a measurement model for the amount of reuse. Examination of the table reveals that:

- There are a small number of citations that address measurement of internal and external reuse (no more 6 citations for any reuse approach).
- There are a small number of citations that address measurement of generation reuse (no more than five citations).

Clearly measurement in the above areas needs to be addressed.

Table 2-14: Breakdown of Reuse Approach

Measurement by Reuse Approach	Number of Citations
Composition	29
External Composition	6
Internal Composition	6
Generation	5
External Generation	1
Internal Generation	0

There are few measurement models for measuring the amount of reuse with analysis and design models. Table 2-5 tabulates the number of citations that demonstrate the use of a measurement model for some kind of software model type based on the stage

of development for the software model type and the reuse approach. For example, a source may describe a measurement model to measure reuse of data flow diagrams. This would count as one citation for measuring analysis composition reuse. Examination of Table 2-5 reveals:

- There are a small number of citations that demonstrate measurement of analysis and design reuse using particular software model types (no more that three sources for any reuse approach).
- There are a small number of citations that demonstrate measurement of generation reuse using particular software model types (no more than four sources for any stage of development).

This further illustrates the need to define a framework that can measure the amount of reuse based on generation reuse, and analysis and design software model types.

Table 2-15: Citations that Demonstrate Measurement of Reuse

Stage and Reuse Approach	Number of Citations
Analysis Composition	3
Analysis External Composition	2
Analysis Internal Composition	1
Design Composition	3
Design External Composition	1
Design Internal Composition	1
Implementation Composition	19
Implementation External Composition	3
Implementation Internal Composition	2
Analysis Generation	1
Analysis External Generation	0
Analysis Internal Generation	0
Design Generation	1
Design External Generation	0
Design Internal Generation	0
Implementation Generation	4
Implementation External Generation	0
Implementation Internal Generation	0

Proposed general approaches to measuring reuse are not adequate

Some more general approaches to reuse measurement could be used, but [228] indicates that these alone will not suffice for measurement of reuse in object-oriented software models. In particular is the work of [18] and [12]. [228] argues that many choices for measurement of object-oriented software models could be made using [12] or [18] and this is not defined by either source. [143] also indicates the problem of measuring reuse with three options available for counting a module. The options are to count the reused module once (reuse level), count the reused module at every occurrence (reuse frequency), and do not count the module at all, since it was not

developed for the current project (ignore external reuse). Each of these options measures something, but how can any of them be chosen, and do any of these options measure what is required?

[18] (p. 157 - 160) also discussed the problems of not being able to derive size measures for specifications and designs easily when using measurement concepts for size with code. For example, data flow diagrams have processes and data stores, how should these be counted? [18] does not recommend pages because they cannot be formally defined but recommends some atomic units to count for size in data flow diagrams, entity-relationship diagrams and a few other modelling paradigms. What is not described is how [18] identified such units, a critical issue identified by [228]. A shift in the development paradigm leads to a change in the structure of the software models, and this requires new measures to be formed for these new software model types and new choices for atomic units. How can these choices be made?

Function points cannot be used because of too many anomalies.

Function points have been suggested as a size measure that can be applied to measurement of the amount of reuse by some authors [17, 28, 211, 215], with [17] even suggesting that it can be used for generation reuse. However, there are two sources worth considering in this respect. Jones [143] describes the problems of using function points as a size estimate in systems projects. Function Point Analysis (FPA) tends to overestimate the amount of code required in these systems projects. Measuring size in this way in the context of the amount of reuse can, therefore, easily result in an overstatement of the amount of reuse occurring. Furthermore, [144] describe how FPA needs to be modified for size estimation both for the domain and the technical environment, particularly the language used. Again, variations of this kind can only further escalate a problem of measurement for the amount of reuse using function points. It is also worth noting that use of a prediction system for size as a measurement system for the amount of reuse has some theoretical conflict. Amount of reuse is a measure, not an estimate, and should be defined as such.

Confusion of Measurement for Modification and Reuse

A number of sources refer to new and changed or modified components in a library model that are applicable to measurement of the amount added and amount reused [25, 213-215, 220, 221, 225, 226, 229-231]. If this happens in the course of reuse it should be known; adequate process definition, data classification, and data collection can identify this. Modification can also be due to other factors such as changing requirements, new product development, or extensions to an application model. The main point is that modification (changing or deleting components in a library model) is not reuse and extension (reusing a library model). By trying to capture

modification and reuse using the same indicator, a critical issue is missed where the activity of modification and activity of reuse is not clearly identified. For example, how do we know whether a low percentage of reuse is due to modification or just poor specification of the component if modification and reuse are merged into one indicator as amount reused or amount added?

[218] (p. 77, 80) gives a basic example of what resolves this dilemma of modification and reuse. Reuse of functions is counted separately from modification (p. 77) but is correlated with functions modified as a consequence of reuse (p. 80). One other source [48] distinguishes between four categories when it comes to measuring size of a software model and calculating productivity. Categories are lines added, lines deleted, lines modified, and lines reused.

Pre-mature validation can have long term consequences

[230, 231] make use of different reuse metrics related to the amount of reuse. This kind of validation study (a study to see if the metrics correlate in some way with benefits) has a purpose but it is a bit like putting the cart before the horse. What if the metrics chosen in the study can't be derived from certain software model types? Another validation study must be conducted on using ones that can. It would be better to first validate some framework for measurement against a variety of software model types.

Now that the main issues have been enumerated, how could a framework be defined that uses meta-modelling to measure the amount of reuse for different kinds of software models? This is the subject of section 2.6.

2.6 Toward Consistent Measurement of Software Reuse

There is a contribution to be made from both sides. On one side, work such as [215] and [228] demand something more specific to a technology. On the other side, work like Frakes [12] and Fenton [18] are trying to set a baseline for measurement that contributes to a consistent collection and interpretation of data for comparison and analysis. Can these two seemingly conflicting views be reconciled? If so how?

Three things are proposed to identify a foundation for measurement of the amount of reuse that can resolve these views.

1. Refinement of meta-modelling as a foundation for structure. It is proposed that meta-modelling is used to make the general framework cope with different kinds of software models.
2. Specification of baseline concepts that describe measurement for the amount of reuse. It is proposed that the framework should contain these concepts as part of its content and structure.
3. Use of set theory. It is proposed that set theory is used to make consistent calculations for measuring the amount of reuse based on the baseline concepts for measurement of the amount of reuse.

Refinement of meta-modelling, the baseline concepts, and set theory for measurement of reuse are described below.

Refinement of Meta-modelling in Software Methodologies

When [18] was published, the issue of changing and variation for software model types appeared to be an insurmountable problem. However, some work in coping with this change on modelling and the development paradigm may shed some light on how this problem can be overcome. The area that shows most promise is meta-modelling. Meta-modelling is chosen because:

1. It is used to provide meta-case tools that overcome the problem of different software model types.

2. Its prominent use in many aspects of software methodologies.

3. Its use in measurement of analysis and design reuse by [27]

1. Meta-case tools can adapt and change part of their functionality to suit different kinds of software models. These tools are based on some kind of meta-model framework. In this way, the same framework is used to model systems using different kinds of software models, and the feasibility of the framework is demonstrated by the meta-case tool. The same meta-case tool is used to automate the modelling process for different kinds of software models. Many examples of such tools can be found in

the literature [163, 166, 171, 173, 193, 232-234]. This suggests that any framework based on meta-modelling should demonstrate its feasibility by automation of the framework.

2. The wide spread use of meta-modelling does suggest it is a useful classification tool. Examples include:

- Software Process Modelling [158, 235, 236].
- Methods Engineering [150, 152, 237].
- Software Measurement [140, 147, 238, 239].
- Requirements Engineering [15, 193, 240].
- Object Technology [35 , 79, 122, 241].
- Software Reusability and Reuse [242 , 243, 244].

3. Meta-modelling was used to define a measurement framework for reuse by [27]. The framework in Banker et al measures the amount of reuse for ten software model types, all of them either analysis or design models. This does suggest that meta-modelling should be used to specify a framework for measuring reuse for numerous software model types.

If meta-modelling is used as a basis for distinguishing clearly the general from the specific in measurement, it may be possible to support a foundation for measurement of the amount of reuse that is specialised into more specific measures for different software model types. However, some refinement of concepts for meta-modelling are necessary to specify any framework for measurement of the amount of reuse based on meta-modelling.

The concept of meta-modelling rarely extends beyond the following description, "A meta-model is a model that describes models". What is unclear is what meta-modelling is, and how it is applied to a problem. Sufficient distinction should be given to illustrate its use rather than just making a model of models or defining in some sense what meta-models should contain for a particular problem. A clear distinction between the essential concepts in meta-modelling and the desirable qualities of meta-models is needed. For example, a class model should be clear and concise (desirable qualities), but a class model always contains classes and relationships (essential concepts). These essential concepts of meta-modelling are used to support specification of meta-models. For example, a class model has classes and relationships and a class model for a hydroponics garden system is an example of a class model.

Many examples of meta-models can be found but few sources can be found that describe concepts of meta-modelling. Therefore, this thesis proposes the following

concepts based on the few sources that provide some contribution to defining essential concepts for meta-modelling. These are:

- **The use of meta-models to describe other meta-models at different levels.** A meta-model is at a given level. A meta-model is a model that describes models, which includes meta-models. Steele and Han [245], based on Nissen et al [193], distinguish between models (m0) that describe a particular application (For example, a warehouse class icon), meta-models (m1) that describe the models used to build particular application models (For example, a class icon that describes class icons including a warehouse class icon), and meta-meta models (m2) that describe meta-models used to build software methodologies (For example, an icon model that describes particular icons including a class icon). This discrimination extends the concept of meta-modelling into multiple levels and provides a good foundation for meta-model instances.
- **Types and Instances.** [152] use the notion of types and instances for relating a higher meta-level to a lower one. Software methodologies are defined by a meta-model at the method level, level 2. For example, the Booch [11] methodology is an instance at level 2. Instances of this meta-model then become types at the next lower level (the application level, level 1). For example, the Booch [11] methodology is a type at meta-level 1. Each software methodology can be used one or more times with a particular path of execution. These are instances of the meta-model for the software methodology and are models at the next lower level (the operational level, level 0). For example, the Booch [11] methodology is used on a given project.
- **The decomposition of meta-models into components at a given level.** [152] also introduce the notion of decomposition or granularity. Granularity is the level of detail at which method fragments function and define software methodologies. For example, a method fragment can define stages (product level), which can be broken down into tasks (model level). These tasks can be broken down into model manipulations (component level). What becomes more apparent is the need to firstly break up a meta-model at the method level into parts called method fragments, and secondly make sure that these components can have differing levels of granularity that affect meta-models of software methodologies and instances of these meta-models that are models at the application level.
- **Each component has a name, a description, and a definition.** [246] is one work that introduces some concepts worth consideration for meta-modelling. [246] address what fundamental concepts should be contained in meta-models at the m2 level or method level. These are the perspective, a system of concepts, and a support or definition of a meta-model as a refinement of its perspective. An

example of perspective is "state charts are used to model states and transitions". A meta-model has a system of concepts that describe it. These have a name and description. For example, the m1 model for a statechart would be states and transitions with their associated descriptions. A support is where concepts that are named are related to each other using some form of systematic definition. For example, in [246] concepts are related to each other using inheritance and are also given attributes to describe them. Thus, a state chart entity can be a state or a transition (relation via inheritance). Transitions have events and actions (attributes).

- **The meta-model architecture as being the sum total of all meta-models used for a particular problem.** [122] and [79] both make use of the term meta-model architecture as a means of describing their different architectures for modelling object-oriented software methodologies. This hints at a distinction between fundamentals of meta-modelling and the meta-model architectures made, with a number of levels separating the different meta-models.

These concepts are refined as part of the proposed framework for measurement of the amount of reuse in Chapter 3.

Baseline Concepts for Measurement of Reuse

It is important to have some basic concepts for measurement of the amount of reuse to provide content to the proposed framework. In this way a consistent basis for measurement is established. These concepts are based on section 2.2 and the literature survey on measurement of the amount of reuse in section 2.5. In the case of generation reuse, concepts used in the research surveying tool are used here because little research has been done to establish any foundation for measurement of generation reuse. The proposed concepts are:

- **Software models and software model types**. A software model is an instance of a software model type. For example, a class model for a car is an instance of the object-oriented class model. (Based on [16, 152])
- **System models and software models**. A system model is composed of a number of software models.
- The *approach to reuse*, either **composition** or **generation**. (Based on [247] and [12]). Composition reuse refers to using of one part of a model in another model. Generation reuse refers to using one model to generate part of ano0ther model.
- The *models used in composition reuse*. These are the **library model** and **application model**. (Based on [215, 225], [212], and [23]). The Library mode is

the model that is reused. The application model is the model that reuses the library model.

- The *models used in generation reuse*. These are the **source model**, the **generated model**, and the **complete model**. (Derived from [215, 225], [212], and [23]). The source model is used to generate the generated model. The generated model is reused in the complete model.
- The *development scope of reuse*, either **internal** or **external**. Internal or external reuse is based on ownership of the reused model. In the case of composition reuse, this refers to the library model. In the case of generation reuse this refers to the source model. Internal reuse is where the library and application, or source and complete models were created within the same project. External reuse is where the library model or the source model were created in different projects from the application model or complete model, respectively (Based on [12] and [18]).
- The *stage of development*. This can be **analysis**, **design**, or **implementation**, but should ***allow for any kind of stage*** of development for reuse. The state of development is attached to each software model. (Based on [18] and [17])
- The *measures for composition reuse*. These are the **amount reused**, **amount added**, and **amount not reused**. (Based on [12], [18], [212], and [23]). For percentage measures these are the **percentage of reuse in the application model**, **percentage added in the application model**, and **percentage of the library model not reused** (Based on [27]).
- The *measures for generation reuse*. These are the **amount generated**, **amount not generated**, and **waste generated** (Converted from [212], [23], and [135]). For percentage measures these are the **percentage contribution to complete model**, **percentage not generated**, and **percentage of waste generated** (Converted from [27]).

This concludes the list of baseline concepts. Now how can set theory be used to provide some structure for measurement of reuse? This is the next subject.

Set Theory for Calculation of the Amount of Reuse

A number of sources in measurement have made use of measures based on sets as part of measurement definition. These include the property based approach to measurement [248], and the Model Order Mapping Approach [249]. Set theory is included as part of defining measures using empirical relation systems and numerical relation systems in one previously cited source [18]. A property based approach is used in two previously cited sources that evaluate the measures for the amount of

reuse [230, 231]. Measures based on sets is used in measurement for object-oriented software models [250, 251]. This includes one previously cited source for measurement of the amount of reuse [212]. On this basis, measures based on sets is used as part of any proposition for measurement of the amount of reuse.

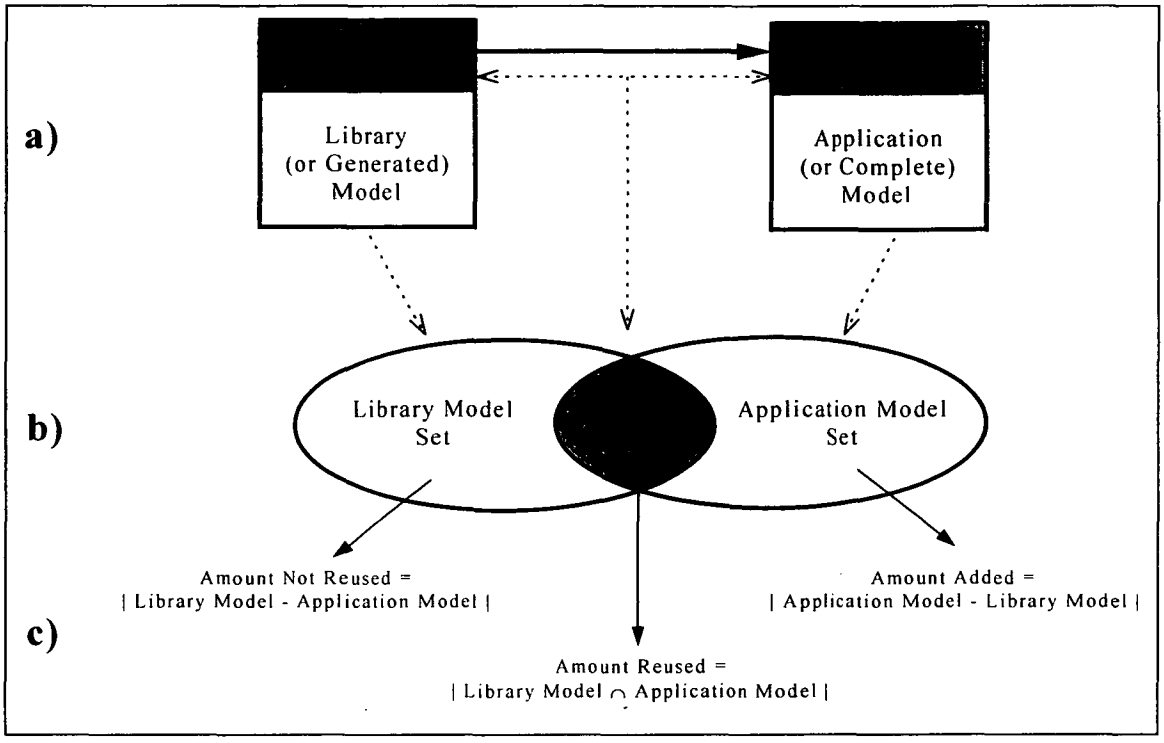

Figure 2-7: Genesis of an Idea for Measures of the Amount of Reuse

Some basis for such a measurement model using measures based on sets needs attention. One diagram in [25] (Figure 37, p. 122) suggests how this could be done (See Figure 2-7). Although this diagram is more complex, three features can be identified in it. First is the boundary of the code of system without reuse. Second is the boundary representing the code of the reused component, part of which overlaps with the code of system without reuse boundary. Third is the pattern representing the code which is not reused. Thus, a common boundary can be identified between the two "Blocks of code" (Figure 2-7a)) as a basis for forming sets based on the models (Figure 2-7b)). If this can be done, then set theory comparison and magnitude operators can be used to measure the amount of reuse (Figure 2-7c)). This approach of comparing two artefacts is also similar to the one used by [211] to measure reuse based on two releases of software. Here are the proposed measures for the amount of reuse based on sets:

- The *measures for composition reuse*. Assume that the library model and application model can be transformed into sets. i.e. Library Model Set and Application Model Set. The measures can be defined as follows:

- The amount reused = | Library Model Set ∩ Application Model Set |
- The amount added = | Application Model Set – Library Model Set |
- The amount not reused = | Application Model Set – Library Model Set |
- The percentage of reuse in the application model
  = (amount reused ÷ | Application Model Set |) * 100
- The percentage added in the application model
  = (amount added ÷ | Application Model Set |) * 100
- The percentage of the library model not reused
  = (amount not reused ÷ | Library Model Set |) * 100

- The *measures for generation reuse*. Assume that generated model and complete model can be transformed into sets. i.e. Generated Model Set and Complete Model Set. The measures can be defined as follows:
  - The amount generated = | Generated Model Set ∩ Complete Model Set |
  - The amount not generated = | Complete Model Set – Generated Model Set |
  - The waste generated = | Generated Model Set – Complete Model Set |
  - The percentage contribution to complete model
    = (amount generated ÷ | Complete Model Set |) * 100
  - The percentage not generated
    = (amount not generated ÷ | Complete Model Set |) * 100
  - The percentage of waste generated
    = (waste generated ÷ | Generated Model Set |) * 100

The astute reader will realize at once that these measures rest on an ambitious assumption. How can a software model be transformed into a set? An even more significant question is how can a framework for measurement of the amount of reuse transform *different software models* based on *different software model types* into sets to measure the amount of reuse? The answer to these questions can be found in Chapter 3. The role of set theory is dealt with in section 3.6.

2.7 Summary

To summarise:

- Software methodologies and software model types have changed in the past and a number of variations exist in the present. UML and object-oriented methodologies are no exception to this.
- The reason for this is the need to change or tailor a software methodology to insure that requirements are met on time and on budget.
- Tailoring of software methodologies suggests a reason for the variation in software model types in the past and present and also suggests a reason for software model types to continue to change in the future.
- Since tailoring of software methodologies appears to be essential, any framework for measurement of the amount of reuse must cope with different kinds of software models, including future software model types.
- As of September 2003, research in measurement of the amount of reuse:
    1. has not adequately addressed the need for a framework to measure reuse for different software model types,
    2. has not demonstrated that any given measurement model for the amount of reuse works for a range of software model types,
    3. has not adequately defined measures for the amount of reuse for UML.
- Therefore, it is proposed that a framework for measurement of the amount of reuse be specified using meta-modelling. This is because meta-modelling was used to specify meta-CASE tools that can cope with different kinds of software models for modelling software systems.
- It is also proposed that a framework for measurement of the amount of reuse be automated and tested using a range of software model types, including those from UML.

Chapter 3 covers how the proposed framework for measurement of the amount of reuse is specified using meta-modelling, and automated.

Chapter 4 describes the experiments that test the proposed framework to see if it can measure the amount of reuse for a range of software model types.

2.8 Sources of Literature

Literature on the topic of software reuse measurement and related issues was obtained from a variety of electronic sources as well as systematic scanning of major conferences and journals. A range of text books on software engineering and software measurement were also included in the study. Electronic sources included Current contents, Carl uncover, and the Collection of Computer Science bibliographies.

Conferences considered for the literature review include the International Conference on Software Engineering; International Conference on Information Systems; Advances in Engineering Software; American Conference on Technology of Object-Oriented Languages and Systems; Australasian Conference on Information Systems; Australian Software Engineering Conference; Conference of the Australasian Cognitive Science Society; ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications; Conference on Software Maintenance; European Conference on Object-Oriented Programming; European Conference on Technology of Object-oriented Languages and Systems; Hawaii International Conference on System Sciences; IEEE International Conference on Software Reuse, IEEE International Software Metrics Symposium; International Symposium on Requirements Engineering; Annual Workshop on Software Reuse; International Conference on Data and Knowledge Systems for Manufacturing & Engineering; International Conference on Deductive and Object-Oriented Databases; International Conference on Information Processing; International Conference on Object-Oriented Information Systems; International Conference on Requirements Engineering; International Conference on Software Engineering: Education & Practice; Pacific Conference on Technology of Object-Oriented Languages and Systems; Proceedings of the IEEE, Symposium on Software Reusability; Symposium on the Foundations of Software Engineering; and the UML International Workshop. Over twenty conferences in total.

Journals considered for the literature review include IBM Systems Journal; Australian Journal of Information Systems; ACM Computing Reviews; ACM Computing Surveys; ACM Transactions on Database Systems; ACM Transactions on Programming Languages and Systems; ACM Transactions on Software Engineering and Methodology; American Programmer; AT&T Technical Journal; Behavioral Science; BT Technology Journal; Communications of the ACM; Computer Language; Computers in Industry; Computing & Control Engineering Journal; European Journal of Operational Research; Expert Systems; IEEE Communications Magazine; IEEE Computer; IEEE Software; IEEE Transactions on Software Engineering; Information and Management; Information and Software Technology;

Information Resources Management Journal; Information Sciences; Information Systems Management; Integrated Computer Aided Engineering; International Journal of Human-Computer Studies; International Journal of Information Management; Journal of Information Systems; Journal of Object-Oriented Programming; Journal of Surveying Engineering; Journal of Systems and Software; MIS Quarterly; Object Magazine; OOPS Messenger; Quality Progress; Sigplan Notices; Software Engineering Journal; Software Engineering Notes; Software Practice and Experience; Systems Practice; and The Computer Journal. Over thirty journals in total.

Chapter 3 Measurement Framework

This chapter has three main goals:

1. To show how the concepts from section 2.6 are incorporated into the measurement framework.
2. To illustrate how the measurement framework measures the amount of reuse for different kinds of software models.
3. To illustrate how the measurement framework is automated.

This chapter satisfies the three goals using seven main sections. Sections 3.2 and 3.3 refine the concepts of meta-modelling from section 2.6 and introduce the measurement framework as a meta-model architecture for measurement. Sections 3.4 – 3.7 elaborate on the most important aspects of the meta-model architecture to illustrate how the measurement framework measures the amount of reuse for different kinds of software models. Section 3.8 describes how the measurement framework is automated.

3.1 Chapter Overview

Let us now consider the chapter 3 in more detail:

3.2 ***Meta-Model Architecture*** refines the concepts of meta-modelling presented in section 2.6 to support specification of meta-model based frameworks. The term meta-model architecture is used to refer to any framework based on meta-modelling. The measurement framework is an example of a meta-model architecture. Specification of a meta-model architecture is done using a meta-level component descriptor (See Appendix A2).

3.3 ***Meta-Model Architecture for Measurement*** shows how all of the concepts from section 2.6 are integrated into a single framework based on meta-modelling. Of particular importance is the description of the static and dynamic part. This is important because it gives an introduction of how the measurement framework can measure the amount of reuse for different kinds of software models.

3.4 ***Overview of Measures*** gives an overview of how set theory (Section 3.6), and baseline concepts for measurement of reuse (Section 3.5) are integrated via model classification (Section 3.7) to support measurement of reuse for different software model types. Model classification acts as a bridge between the need to capture qualitative data such as the names of software models (Section 3.5), and quantitative data such as measures for the amount reused (Section 3.6). This section also identifies the more important aspects of the measurement framework by showing

which components in the meta-model architecture are responsible for data classification (Section 3.5), set theory (Section 3.6), and model classification (Section 3.7).

3.5 ***Data Classification*** illustrates how the baseline concepts for measurement of reuse from section 2.6 are integrated into the measurement framework.

3.6 ***Set Theory*** illustrates how set theory from section 2.6 is used to measure the amount of reuse for different kinds of software models. When software models are classified as software model sets it is a simple matter of using set operators to calculate the amount of reuse. The magnitude operator is usually combined with either the difference or intersection operators to support measurement of the amount of reuse.

3.7 ***Model Classification*** illustrates two things. Firstly it shows how software models are classified to support their transformation into software model sets for measurement of the amount of reuse. Secondly, section 3.7 revisits the static and dynamic part to elaborate on how different kinds of software models are classified to support measurement of reuse.

3.8 ***Automation Specification*** describes how the significant components in the meta-model architecture are implemented in the prototype tool. This is done to show how the measurement framework is automated.

3.2 Meta-Model Architecture

In this thesis, the concepts for meta-modelling are used as a foundation for a more general approach to measuring the amount of reuse. This foundation is analogous to a requirements specification for measurement of reuse. By itself it is not enough to implement a tool for measurement, but it provides a foundation for one. The addition to set theory and UML specifications for software models is indicative of the extensions required for automation. Although meta-modelling is popular, automation of it is still limited. Few sources approach the level of automation achieved in this thesis. To summarise, meta-models are defined at a given meta-level. A meta-model contains a number of meta-level components at the same meta-level. A meta-level component is defined using one or more definition methods. These are also meta-level components. A meta-level component at its own level is a type with instances. These become types at the next lower meta-level. Meta-level components can have dependencies between each other at the same meta-level. This is based on definition methods for meta-level components.

Fundamentals of Meta-Modelling

To specify a framework for measurement, some meta-model concepts for specifying a meta-model architecture are introduced by detailing the links between meta-model levels and enumerating the qualities of meta-models. Meta-modelling is divided into core concepts and extensions. Core concepts are deemed essential for meta-modelling. Extensions are not required but provide some refinements to meta-modelling.

Core Concepts

Meta-Level Component (MLC): A meta-model architecture has a number of named components called meta-level components (MLCs). For example, the MLC named Software Model Type Classification in Figure 3-1.

Meta-Level (MetLn): A meta-level component is at a given meta-level (MetLn or mn; n is an integer). For example, the Software Model Type Classification MLC is at MetL2.

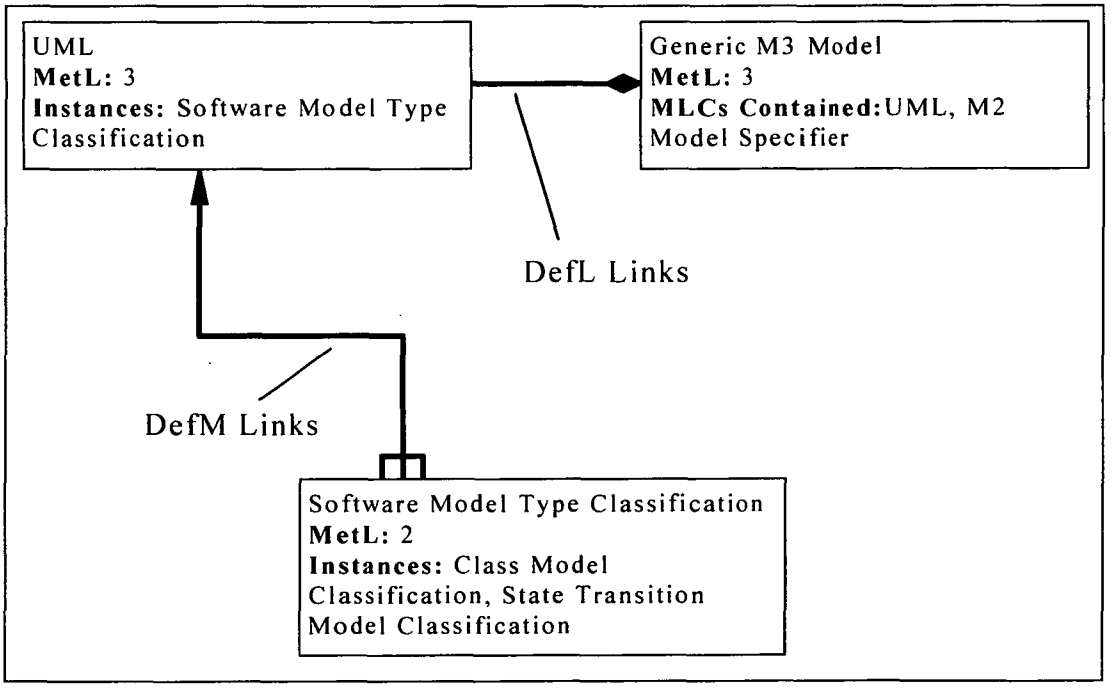

Figure 3-1: *Examples of MLCs*

Interpretation or Meaning (IMea): A meta-level component has some kind of interpretation or meaning in a model based on the level (IMea). For example, in Figure 3-1 the interpretation for the Software Model Type Classification MLC could be "The Software Model Type Classification MLC is used to classify different kinds of software models. For example, a class model in object-oriented modelling or a data flow model from structured process modelling can be classified using this MLC.

Thus, instances of this MLC can include a class model, a data flow model, or a state transition model".

Type and Instance: An MLC is a type with instances of itself at its own level, which become types or MLCs at the next lower meta-level. For example, at meta-level 2, the Software Model Type Classification has instances such as the Class Model Classification in Figure 3-1. At meta-level 1 the Class Model Classification becomes a MetL1 MLC as part of a software methodology. MetL1 MLCs describe models of some kind, but not MLCs.

mn Model (or n*meta-model): The sum of MLCs at a given MetLn constitute a *n*meta model* or *mn model*. For example, the UML and Generic M3 Model MLCs are part of a meta-meta-meta model or m3 model in Figure 3-1. It is possible for a number of meta-models to exist at a given meta-level. For example, Figure 3-1 in the class model classification and state transition model classification are meta-models or m1 models. In this case the sum of meta-models at a given meta-level constitutes a meta-model layer.

Definition Method (DefM): An MLC's meaning is enhanced if it has one or more definition methods (DefM) that systematically describe it to some degree, which are MLCs at one metal-level higher than itself. For example, the Software Model Type Classification is defined using the UML MLC in Figure 3-1. Note that an MLC can act as a definition method (DefM) for MLC at different meta-levels. This implies that the MLC can be member of different meta-models (mn models) at different meta-levels (MetLn).

Base Definition Method (Base DefM): MLCs that do not have definition methods are referred to as base DefMs. For example, the EBNF variation (TDL) at meta-level two is used to classify programming languages at meta-level one, but may not be defined using any other MLCs. MLCs that are defined using themselves can imply that an MLC of the same kind exists at one meta-level higher than the defined MLC. For example, TDL is defined using TDL. This implies that the TDL MLC exists at meta-level two and three. The DefM attribute enhances the specification of a meta-model architecture between levels by using the DefM links from lower level MLC's to higher ones to determine the MetLn of an MLC.

Extensions

Definition Location (DefL) and Base Definition Location (Base DefL): If there are a number of MLCs at a given MetLn, then an MLC may have a definition location in a meta-model (DefL) that is usually an MLC at the same meta-level. For example, the definition of the Software Model Type Classification is located in the

Measurement m2 Model in Figure 3-1. The DefL attributes enhance the meta-model architecture specification by grouping different MLCs at the same level into a n*meta-model. MLCs that do not have a definition location are referred to as base DefLs.

MLC Description (MLCDesc): An MLC may describe one or more MLCs at one meta-level lower than itself, the lowest being MetL1 (MLCDesc); this is a reverse of the DefM attribute.

MLC Dependency (MLCDep): At each meta-level there are relationships or dependencies that link MLCs together (MLCDep). Dependency types include prerequisite or existence dependency (an MLC instance depends on another MLC instance before defining itself), aggregate dependency (an MLC instance is part of or composed of other MLC instances), and classification dependency (one MLC instance is a more specific kind of another MLC instance). The dependencies are similar to the three basic associations in object-oriented modelling, namely Using or Communication, Aggregation, and Inheritance, respectively.

The MetLn, DefM, DefL and IMea attributes of an MLC are a foundation for establishing and enhancing MLCDep between MLCs at the same meta-level (MetLn) and MLCDesc between MLCs at adjacent meta-levels. MLCDep and IMea qualities are the most flexible and least specific part of a meta-model architecture.

3.3 Meta-Model Architecture for Measurement

The meta-model architecture for measurement of the amount of reuse is detailed in Appendix A2. The architecture has four main layers. The section under the heading "Overview" covers the more significant parts of the architecture that address the requirement for measurement of different software model types. Following this is a brief description of the layers and the use of different MLCs in the meta-model architecture. To summarise:

- The M4 and M3 layers specify a baseline for a generic meta-model architecture that provides classification tools for the M2 and M1 layers. These classification tools are represented as MLCs at meta-level 4 and 3 that act as definition methods for MLCs at meta-level 2 and 1.
- The M2 and M1 layers specify the remainder of the architecture for measurement of the amount of reuse. These layers are primarily responsible for measuring the amount of reuse for a different kinds of software models.
- All layers (M4, M3, M2, M1) also have an MLC to ensure that the fundamentals of meta-modelling defined in section 3.2 are used to specify MLCs.

Overview

Figure 3-2 illustrates the meta-model architecture for measurement of the amount of reuse. This figure illustrates how the dynamic part of the meta-model architecture can support measurement of the amount of reuse for different software model types.

At meta-level four UML and TDL are used to define MLCs at lower meta-levels.

At meta-level three UML is defined using itself and Set Theory is defined using TDL. Both UML and Set Theory are used to define a classification scheme for various software model types.

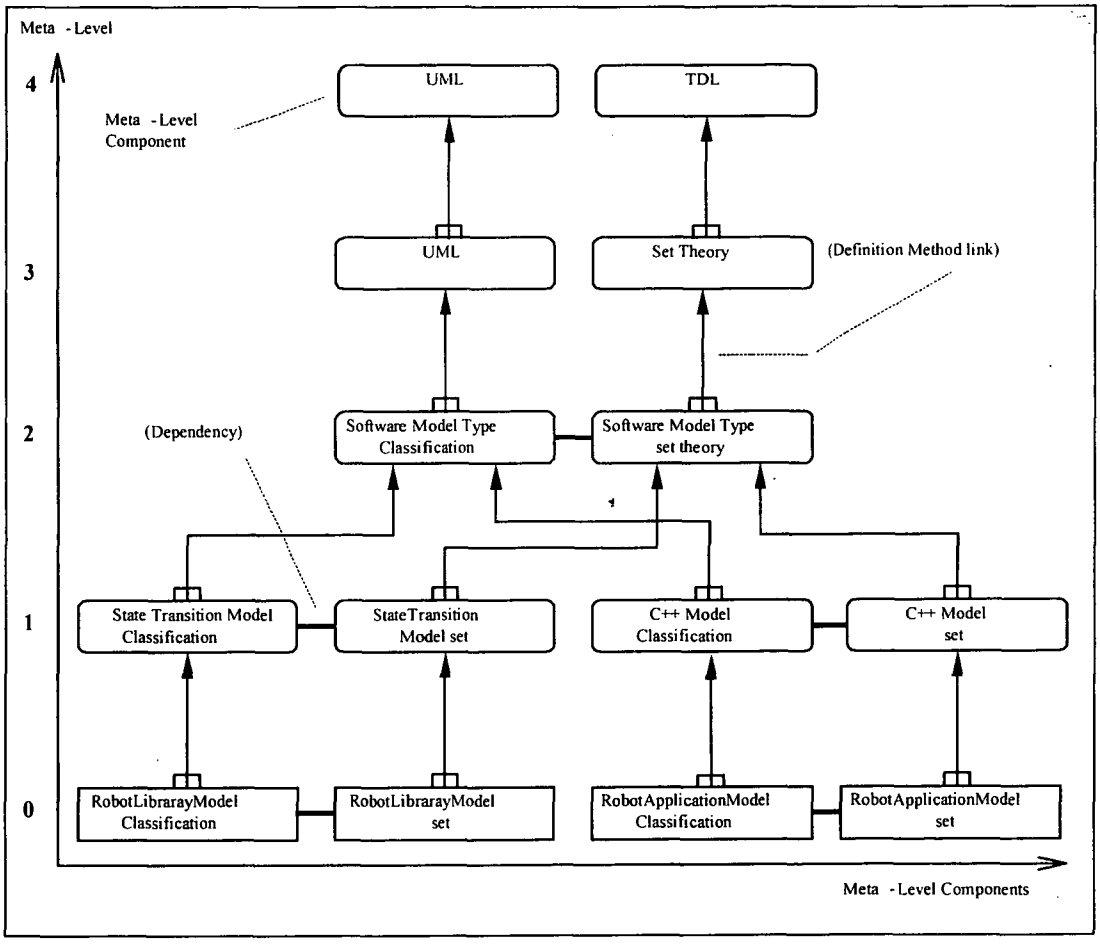

Figure 3-2: *Meta Model Architecture of Measurement Framework*

At meta-level two meta-level components represent the classification scheme for classifying software model types to measure the amount of reuse. The Software Model Type Classification and Software Model Type Set Theory meta-level components are defined using UML and set theory (Instances of UML and Set Theory MLCs). The Software Model Type Classification and Software Model Type Set Theory MLCs classify various software model types in a way suitable for measurement of the amount of reuse. For example, the software model type named C++ Model is defined using the Software Model Type Classification meta-level

component. This allows for measurement of the amount of reuse for different kinds of software models by adding instances of the Software Model Type Classification and Software Model Type Set Theory MLCs.

At meta-level one meta-level components represent various software model types. These are defined using the Software Model Type Classification and Software Model Type Set Theory MLCs. MLCs at meta-level two define software models based on their type. For example, the software model type named C++ Model Classification is a meta-level component. The C++ Model Classification can have instances of the C++ Model at meta-level zero.

At meta-level zero (The model level) are the various software models. These are not meta-level components, and are defined using the meta-level components at meta-level one. For example, the software model named Robot Complete Model is defined using the C++ Model Classification meta-level component and C++ Model set meta-level component.

Static and Dynamic Part

It is important to draw a distinction between the static part and the dynamic part of the meta-model architecture. The static part of the architecture is represented by the MLCs described for each layer of the meta-model architecture. These MLCs do not change. The dynamic part of the meta-model architecture is illustrated in Figure 3-2 as meta-level one MLCs and their instances. For example, the state transition model set MLC and its instances, the RobotLibraryModel set and RobotApplicationModel set represent the dynamic part of the meta-model architecture. New dynamic MLCs and Instances are added at meta-level 1 to measure the amount of reuse for different kinds of software models. For example, to measure the amount of reuse for data flow models, a dynamic data flow model classification MLC and a dynamic data flow model set MLC are added at meta-level 1. Then instances of these MLCs are added at meta-level 0 to measure the amount of reuse.

Meta-Model Diagrams

To give an overall picture of a given meta-model architecture, two diagrams are used. The notation used is UML. The first diagram is called the meta-model architecture diagram. This diagram illustrates the layers in the meta-model architecture giving their level for the layer and the name. This diagram is represented using one Package per layer, this layers at a lower meta-level pointing to packages at a higher meta-level. More specific packages within the "layer" package can also be illustrated. These are referred to as partitions and are identified using the stereo type <<Partition>>. The layer package is indicated using the stereo type <<Mn Layer>>,

where n is the meta-level of the layer. The diagram is referred to as the meta-model architecture diagram. An example is given in Figure 3-3.

The second diagram is called a meta-model layer diagram. This diagram details the meta-level components in a layer or a partition within a layer. MLCs are represented using classes. MLC's that are used as definition methods (DefMs) are pointed to using an arrow with a pitchfork tail. The tail is attached to the lower level MLC and points to the higher level MLC. At least one meta-model layer diagram exists for each meta-model layer. MLCs that are used as definition locations (DefLs) for other MLCs use the aggregation symbol with the diamond filled. The MLC that is the DefL for another MLC is the aggregate, the other MLC is the component. An example is given in Figure 3-4. Figure 3-3 shows the meta-model architecture diagram for the measurement framework.

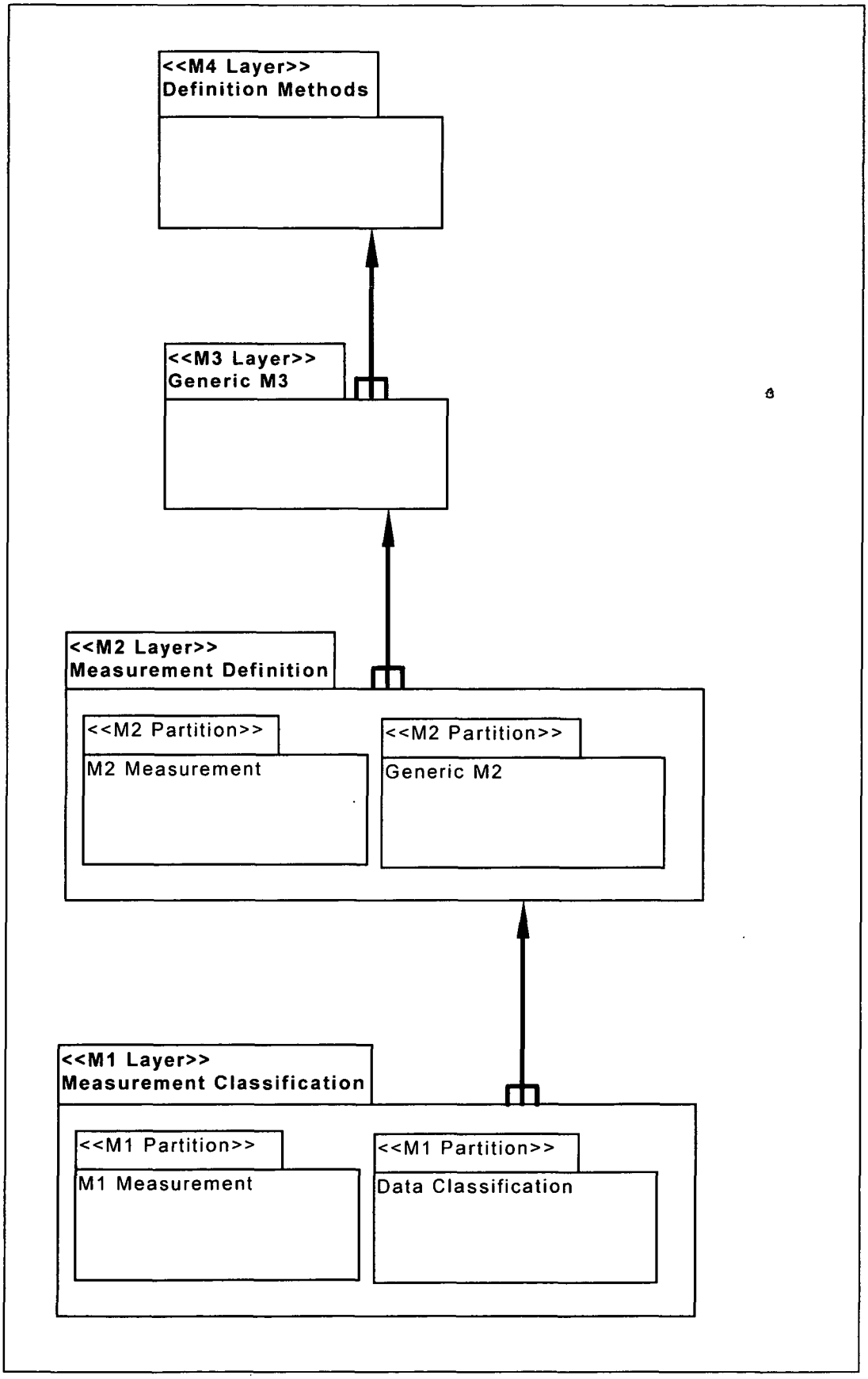

Figure 3-3: *Meta-Model Architecture Diagram*

Overview of Layers

The meta-model architecture has four main categories of MLCs. Table 3-1 identifies the MLCs in the meta-model architecture for each category.

The first category MLCs are used to ensure that the fundamentals of meta-modelling from section 3.1 are used to define the meta-model architecture. For example, the Measurement Model Specifier at meta-level 3 ensures that MLCs at meta-level 2 are specified using the fundamentals of meta-modelling from section 3.1.

Table 3-1: Categories of MLCs at each layer

Layer	First Category	Second Category	Third Category	Fourth Category
The Meta-Level of the MLC	MLCs used to enforce fundamentals of meta-modelling	MLCs used a definition methods	MLCs used to measure the amount of reuse for a range of software model types	MLCs used to measure the amount of reuse for specific software model types
4	M3 Model Specifier	Text Definition Language (TDL)		
3	Measurement Model Specifier	Generic M3 Model TDL UML Set Theory		
2	Amount of Reuse Measurement M1 Model Specifier	Amount of Reuse Measurement M2 Model TDL UML Set Theory	Software Model Type Classification Software Model Type Set Theory	
1	Amount of Reuse Measurement Model Specifier		Software Model Classification Software Model Set Theory Context Data Classification Measurement Data Classification	State Transition Model Classification State Transition Model set C++ Model Classification C++ Model set
0				RobotApplicationModel Classification RobotApplicationModel set

The second category of MLCs are used to define other MLC's. i.e. they act as definition methods for lower level MLCs. For example, the Set Theory MLC at meta-level 2 is used to define the Software Model Type Set Theory MLC at meta-level 1.

The third category of MLCs are used to measure the amount of reuse for a range of software model types. These MLCs define the structure of dynamic MLCs by acting as definition methds for dynamic MLCs. For example, the Software Model Set Theory MLC at meta-level 1 is used to specify software models as sets for measuring the amount o reuse.

The fourth category of MLCs are defined using the third category of MLCs to measure the amount of reuse for specific software model types and specific software models. For example, the C++ Model set MLC at meta-level 1 is used to define the RobotApplicationModel set at meta-level 0.

The first, second, and third category of MLCs form the static part of the meta-model architecture. The fourth category of MLCs form the dynamic part of the meta-model architecture.

M4 Layer

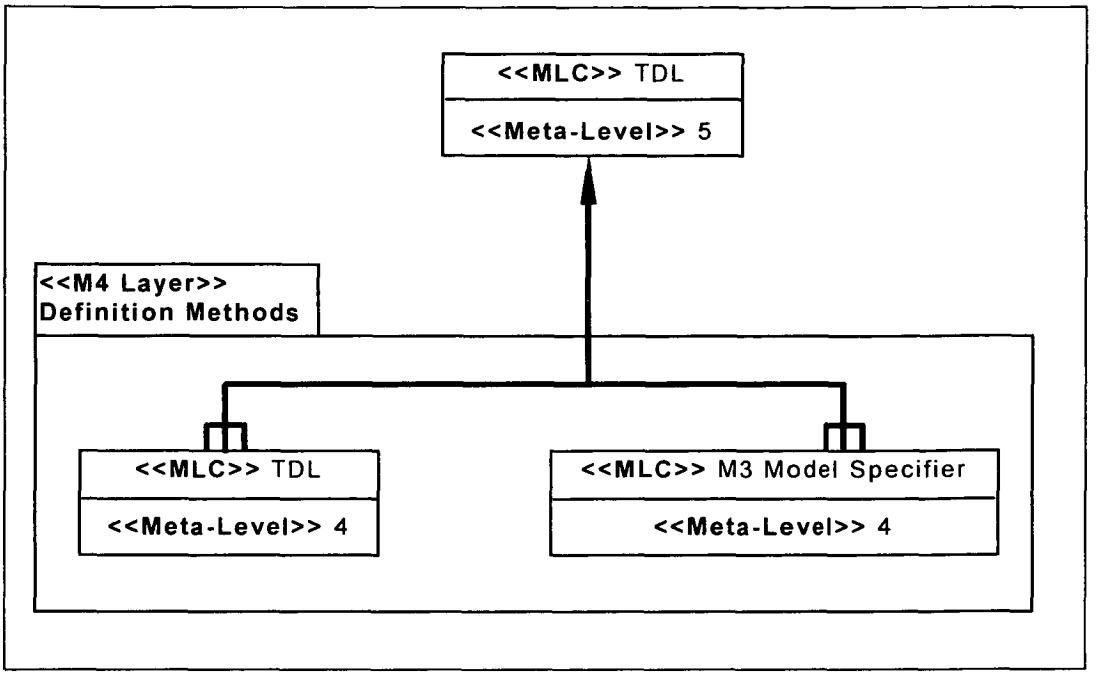

Figure 3-4: *M4 Layer Diagram*

There are two MLCs at meta-level four. These are:

1. **Text Definition Language** (TDL). TDL stands for Text-Definition Language. TDL is a variation of Extended Bakus-Naur form for defining the syntax of text languages [1-6]. TDL is used as a definition method to structure meta-level 3 MLCs, for example, TDL is used to specify the Set Theory MLC.
2. **M3 Model Specifier**. This MLC is used to structure meta-level 3 MLCs according to the fundamentals of meta-modelling described in section 3.2 and using the MLC descriptor described in Appendix A2. In this thesis the M3 Model Specifier is used to specify the Generic M3 Model MLC at meta-level 3. This is done to ensure that the concepts of meta-modelling defined in section 3.2 are used to specify the meta-model architecture for measurement.

M3 Layer

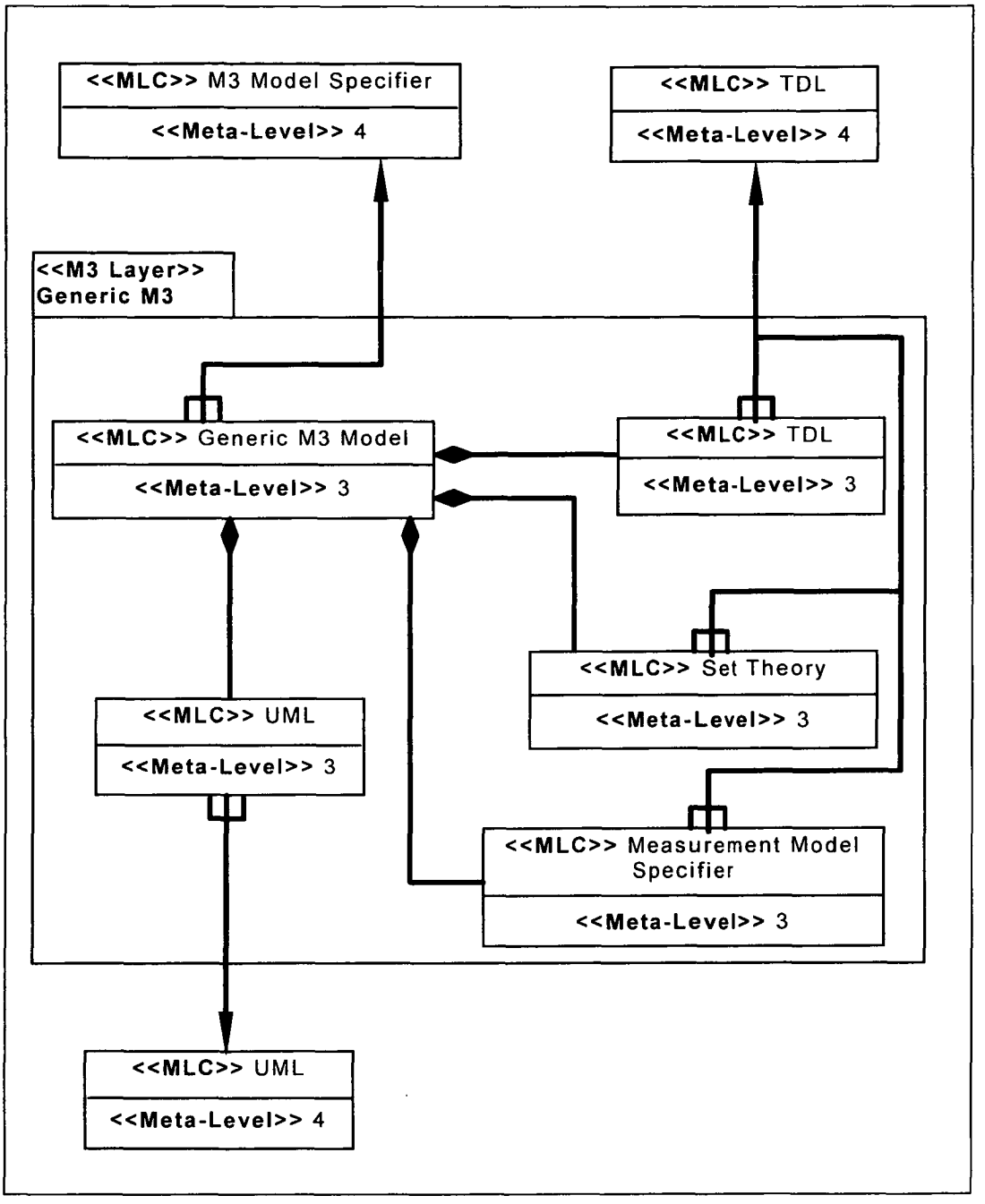

Figure 3-5: *M3 Layer Diagram*

There are five MLCs at meta-level three. These are:

1. **Generic M3 Model**. This MLC contains all the MLCs used as definition methods for lower level MLCs. It is defined using the M3 Model Specifier MLC. The remaining MLCs at this level are contained in the Generic M3 model.
2. **UML**. This MLC is used as a definition method for meta-level 2 MLCs that are defined using UML. For example, the Software Model Type Classification MLC at meta-level 2 is defined using UML.

3. **Set Theory**. This MLC is used as a definition method for meta-level 2 MLCs that are defined using mathematical set theory. For example, the Software Model Type Set Theory MLC at meta-level 2 is defined using Set Theory.
4. **TDL**. TDL is used as a definition method for meta-level 2 MLCs that are defined using text language syntax. This is essentially the same as the TDL at meta-level 4, but needs to be re-specified at this level.
5. **Measurement Model Specifier**. This MLC is used to specify meta-level 2 MLCs that are used for software measurement according to the fundamentals of meta-modelling described in section 3.2 and using the MLC descriptor described in Appendix A2. For example, the Amount of Reuse Measurement M2 Model MLC is an instance of the Measurement Model Specifier. The definition methods for these MLCs are either UML, Set Theory, or TDL. This MLC is defined to ensure that the concepts of meta-modelling defined in section 3.2 are used to specify the meta-model architecture for measurement.

TDL, UML, and Set Theory are referenced as definition methods at different levels, implying that these MLCs also exist at different levels. This must be done so that they can be used as definition methods at different levels.

M2 Layer

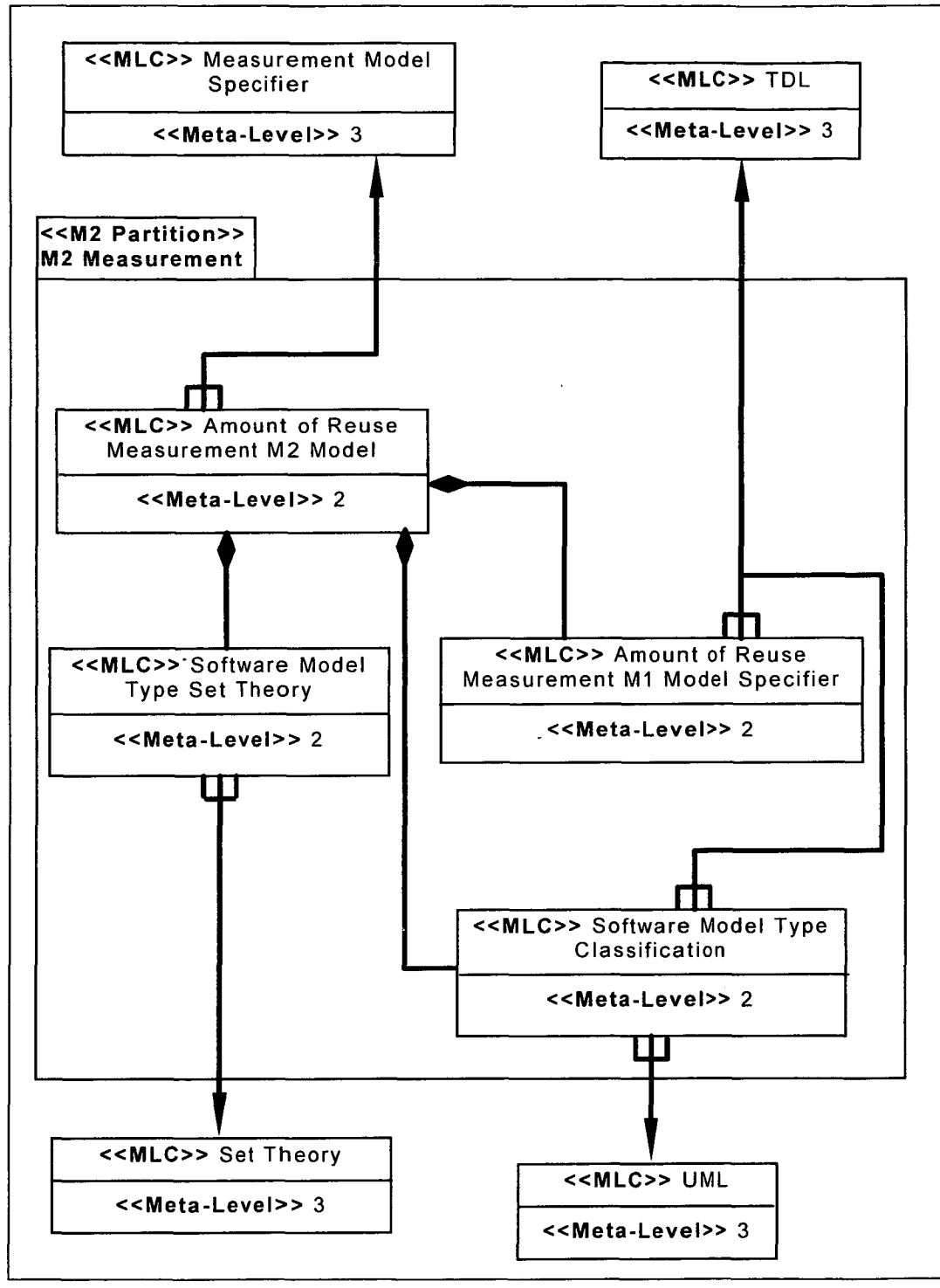

Figure 3-6: *M2 Layer, M2 Measurement Partition*

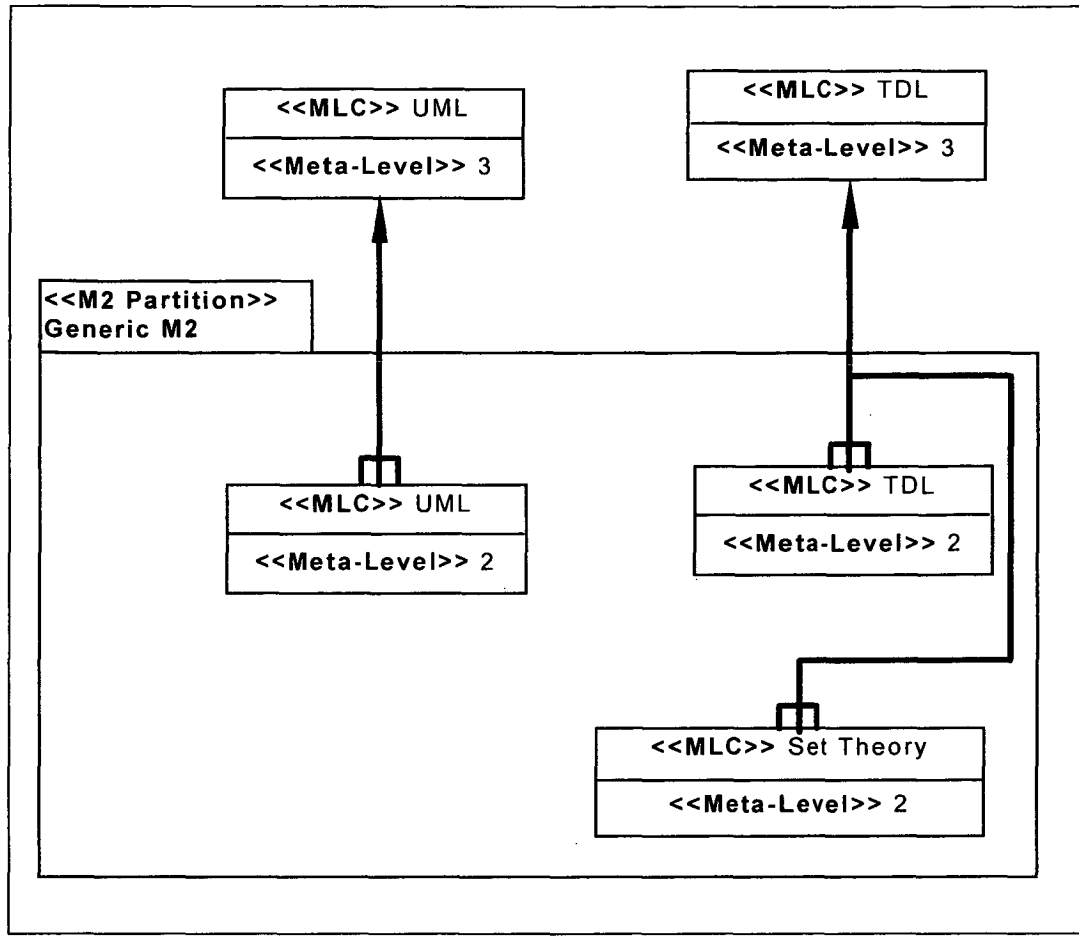

Figure 3-7: *M2 Layer, Generic M2 Partition*

There are seven MLCs at meta-level two. These are:

1. **TDL**. This MLC is used as a definition method for lower level MLCs. It is the same as the TDL MLC at meta-level 4.
2. **UML**. This MLC is used as a definition method for lower level MLCs. It is the same as the UML MLC at meta level 3.
3. **Set Theory**. This MLC is used as a definition method for meta-level 1 MLCs that are defined using mathematical set theory. For example, the Software Model Set Theory MLC at meta-level 1 is defined using Set Theory.
4. **Amount of Reuse Measurement M2 Model**. This MLC is defined using the Measurement Model Specifier MLC. The Amount of Reuse Measurement M2 Model MLC is used to classify different kinds of software models for the purposes of measurement of the amount of reuse. The remaining MLCs at this level are contained in the Amount of Reuse M2 Model.
5. **Software Model Type Classification**. This MLC is defined using UML and TDL. The Software Model Type Classification MLC is used to classify different kinds of software models. For example, in Figure 3-2 the C++ Model

Classification MLC is an instance of the Software Model Type Classification MLC. Aspects of this MLC are described in more detail in section 3.7 to illustrate how the measurement framework classifies different kinds of software models to support measurement of the amount of reuse.

6. **Software Model Type Set Theory**. This MLC is defined using Set Theory. The Set Theory MLC is used to generate a set of model element types based on the software model type classification. For example, in Figure 3-2 the C++ Model set MLC is an instance of the Software Model Type Set Theory MLC. Aspects of this MLC are described in more detail in section 3.6 to illustrate how the measurement framework uses set theory to classify different kinds of software models for measurement of the amount of reuse.

7. **Amount of Reuse Measurement M1 Model Specifier**. This MLC is defined using TDL. This MLC is used to group M1 models for measurement of the amount of reuse based on the Software Model Type Classification and Software Model Type Set Theory MLCs and according to the fundamentals of meta-modelling described in section 3.2. For example, in Figure 3-2 C++ Model Classification MLC and C++ Model Set MLC are contained in a C++ Amount of Reuse Measurement M1 Model MLC. This MLC is defined to ensure that the concepts of meta-modelling defined in section 3.2 are used to specify the meta-model architecture for measurement.

M1 Layer

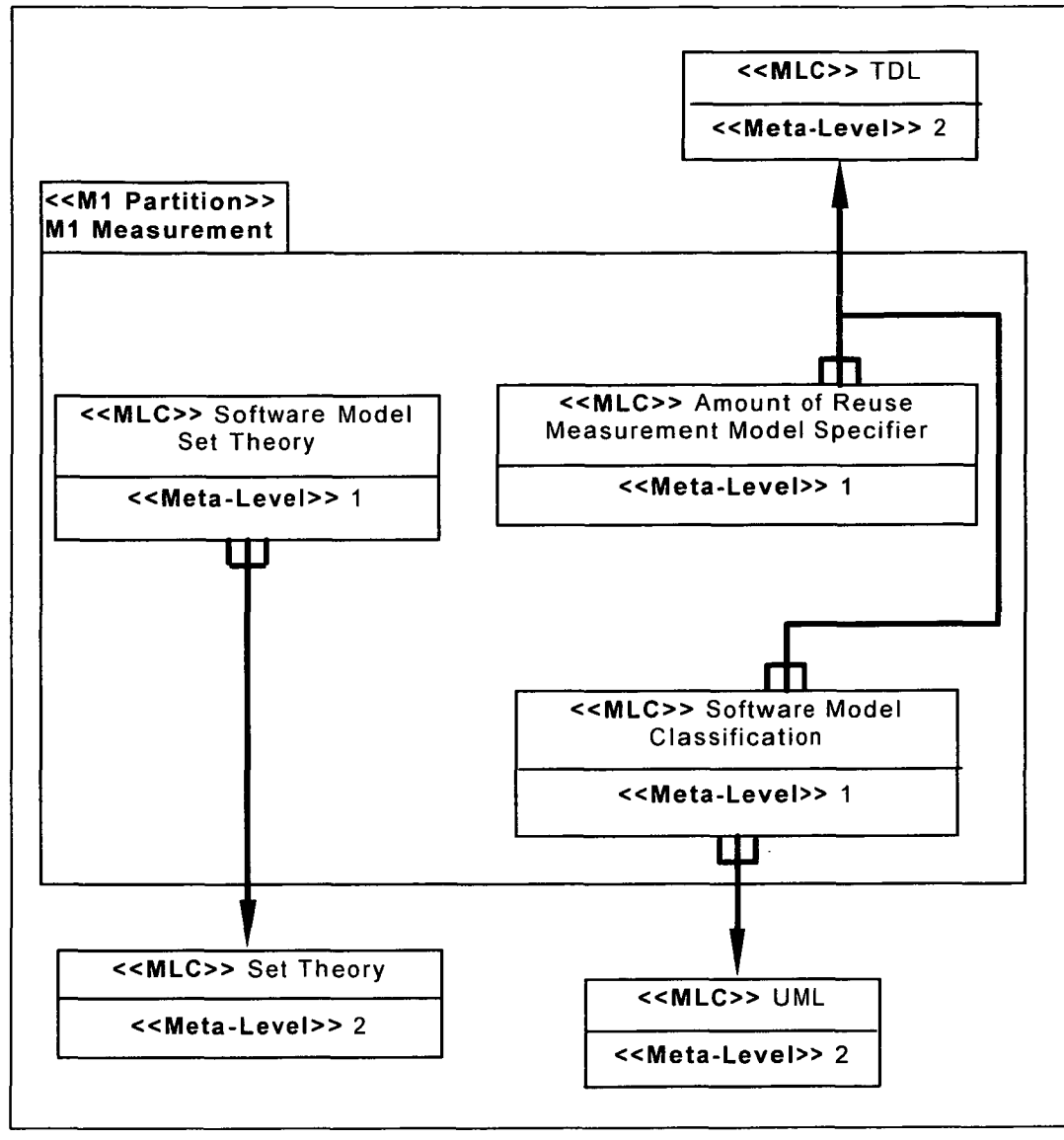

Figure 3-8: *M1 Layer, M1 Measurement Partition*

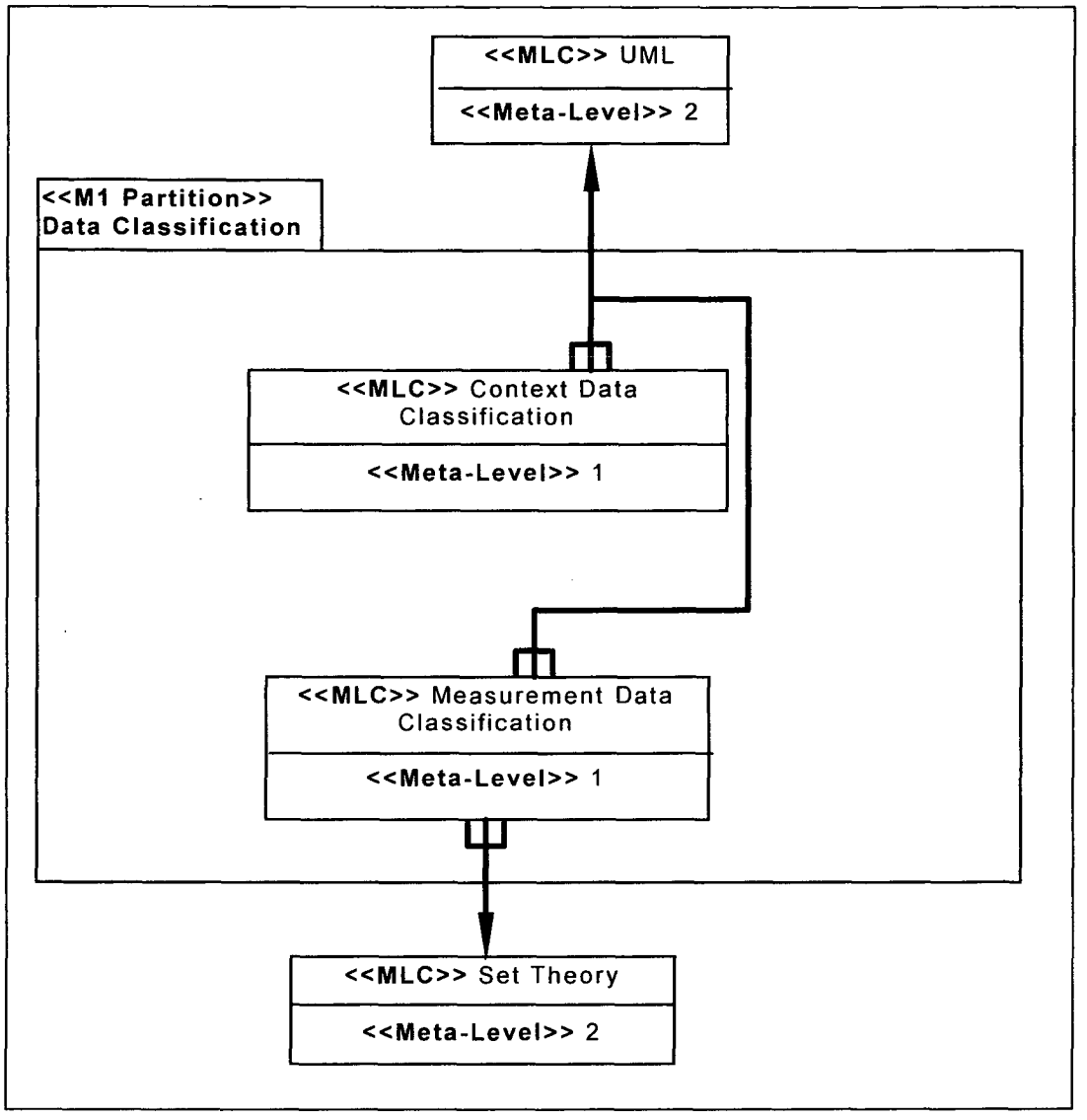

Figure 3-9: *M1 Layer, Data Classification Partition*

There are five MLCs at meta-level one. These are:

1. **Software Model Classification**. This MLC is specified using UML and TDL. The Software Model Classification MLC is used to classify software models based on their software model type classification. For example, in Figure 3-2 the RobotApplicationModel Classification is an instance of the Software Model Classification MLC. Aspects of this MLC are described in more detail in section 3.7 to illustrate how different software models are classified based on their software model type to support measurement of the amount of reuse.

2. **Software Model Set Theory**. This MLC is specified using Set Theory. The Software Model Set Theory MLC is used to generate software models as sets based on their software model type set. For example, in Figure 3-2 the RobotApplicationModel set is an instance of the Software Model Set. The RobotApplicationModel set is based on the C++ Model set. Aspects of the MLC

are described in more detail in section 3.6 to illustrate how set theory is used to obtain measures for the amount of reuse for different software models based on their software model type.

3. **Amount of Reuse Measurement Model Specifier**. This MLC is specified using TDL. The Amount of Reuse Measurement Model Specifier is used to group software models specified using the Software Model Set Theory and Software Model Classification MLCs and according to the fundamentals of meta-modelling described in section 3.2 and using the MLC descriptor described in Appendix A2. For example, in Figure 3-2 the RobotApplicationModel classification and RobotApplicationModel set are contained in the RobotApplicationModel Amount of Reuse Measurement Model. This MLC is defined to ensure that the concepts of meta-modelling defined in section 3.2 are used to specify the meta-model architecture for measurement.
4. **Context Data Classification**. This MLC is defined using UML. The Context Data Classification MLC is used to classify qualitative data. i.e. this MLC specifies the context of the measures. Aspects of this MLC are described in more detail in section 3.5 to illustrate how baseline concepts for measurement of reuse from section 2.8 are incorporated into the measurement framework.
5. **Measurement Data Classification**. This MLC is defined using UML. The Measurement Data Classification MLC is used to group measurement data for the amount of reuse and relate it to qualitative data for collection and analysis. Aspects of this MLC are described in more detail in section 3.5 to illustrate how baseline concepts for measurement of reuse from section 2.8 are incorporated into the measurement framework.

There is also a meta-level one MLC defined for each software model type. These MLCs are defined using the Amount of Reuse Measurement M1 Model Specifier. For example, C++ Amount of Reuse Measurement M1 Model MLC. These MLCs are the dynamic part of the architecture and not the static part defined here.

M0 Layer

Instances of the Amount of Reuse Measurement Model Specifier MLC exist at this level, one for each software model. Instances of the Context Data Classification and Measurement Data Classification MLC also exist at this level. For example, the RobotApplicationModel Amount of Reuse Measurement Model. MLCs do not exist at meta-level 0 and this is why they are not described here.

3.4 Overview of Measures

Now that we have a meta-model architecture for measurement of the amount of reuse it is time to elaborate on those key aspects that demonstrate how the measurement framework measures the amount of reuse for different kinds of software models. This is the purpose of sections 3.4 to 3.7. Table 3-2 links the most significant MLCs from the meta-model architecture for measurement to sections 3.5 to 3.7 to highlight the key aspects of the measurement framework chiefly responsible for measuring the amount of reuse for different kinds of software models.

Table 3-2: Relationship between MLCs and Sections 3.5 – 3.7

MLC Name	Definition Method	Section	Sub-Heading(s)
Context Data Classification	UML	3.5	Fundamentals
Measurement Data Classification	UML	3.5	Composition Reuse Generation Reuse
	Set Theory	3.6	Composition Reuse Generation Reuse
Software Model Type Set Theory	Set Theory	3.6	Fundamentals
Software Model Set Theory	Set Theory	3.6	Fundamentals
Software Model Type Classification	UML	3.7	Fundamentals
	TDL	3.7	Representation of Software Model Types
Software Model Classification	UML	3.7	Fundamentals
	TDL	3.7	Representation of Software Models

In this thesis, measures are characterised from three views. Firstly, there is the *qualitative* side of the measure. This means the context and place to which the measure is applicable for data collection and analysis. Qualitative variables are described using UML in section 3.5 These models are put forward as the means by which the data for measures is collected, analysed, and compared, i.e. they act as triangulation variables.

Secondly, there is the *quantitative* side. Here there is a need to characterise and constrain the measure in a suitable form for making some numerical calculation. Set theory is used to characterise the nature of calculation of a given measure. Quantitative variables are described in section 3.6.

Thirdly, there is the need to characterise the calculation based on some qualitative analysis of the thing measured. This is referred to as the *linkage* side. In this work UML is used to describe the linkage in section 3.7. The linkage was achieved using a

model that characterises the structure of different software model types. The linkage acts as a kind of bridge between the qualitative side and the quantitative side between the measures and the software models. Qualitative because the software model type describes the structure of software models. Quantitative because instances of the given model are used to support calculations for measurement of the amount of reuse based on the set theory.

3.5 Data Classification

Fundamentals

The model in Figure 3-10 illustrates the relationship between software models and software development. This model is from the Context Data Classification MLC at meta-level 1. The model is used to analyse and characterise the data for both composition and generation reuse. External reuse is defined by comparing the membership of software models to software projects. When both software models are from the same project, then an instance of internal reuse is measured. When software models are members of different software projects, a measure of external reuse is measured. Software models in libraries can be part of a library project. Software models are still developed by software developers using implementations (For example, the Paradigm Plus® CASE tool) and this is manifest in the "Made by" and Aggregate relationships.

A software project is described by a number of system models. Each system model represents an entire software application at a given level of abstraction. A system model is composed of a number of software models. Each software model describes part of the system model. A software model is an instance of a software model type. A software model type is automated on one or more implementations. A software model is located in one or more implementations. A software model has a date stamp signifying when the software model was completed. A software model type is based on one or more literature sources that contribute to knowledge about the software model type and its application to software methodologies.

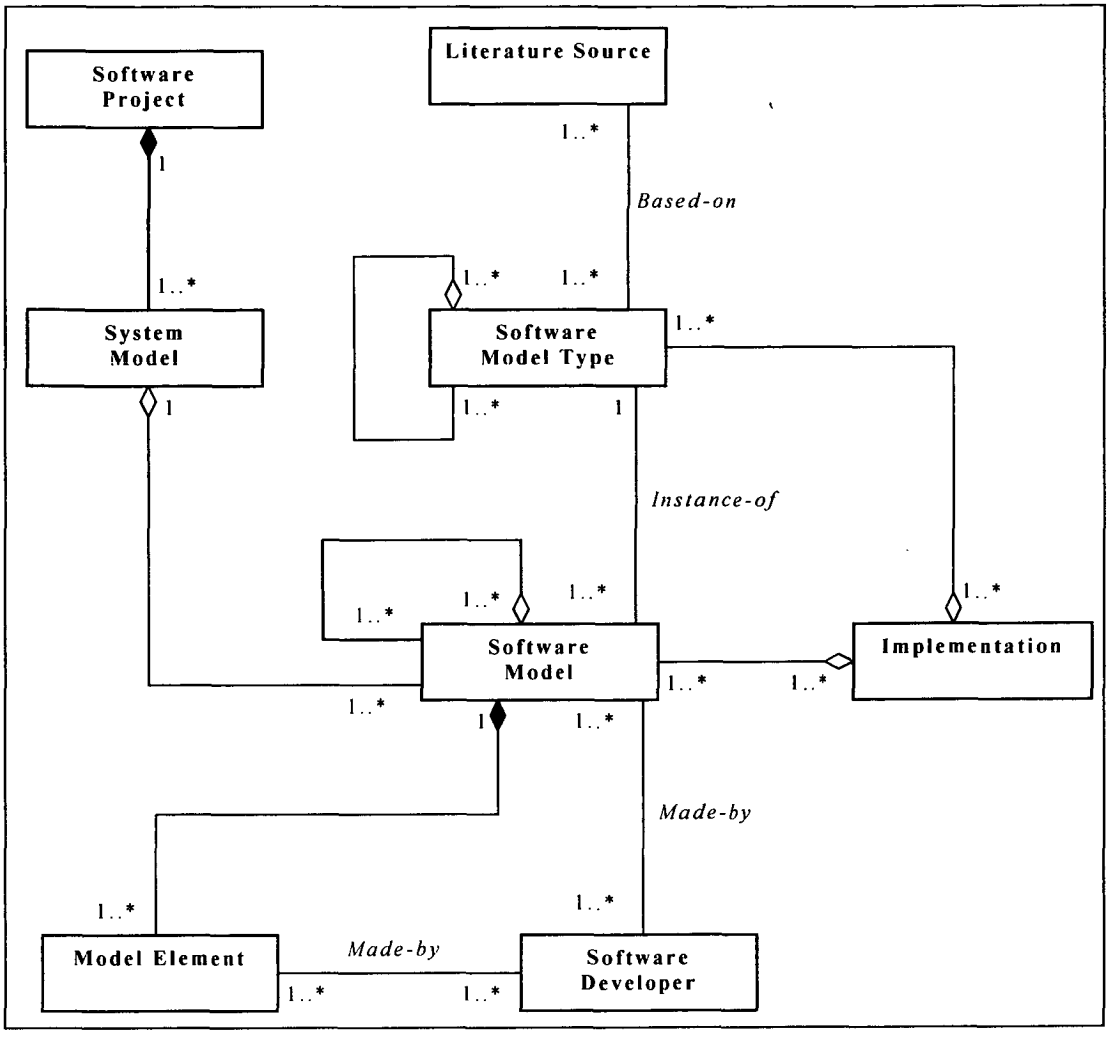

Figure 3-10: *Model for Classification of Context Data*

A worked example: A software project can be an automotive dashboard application. System models for the software project include the automotive dashboard analysis model and the automotive dashboard design model. The words "analysis" and "design" indicate the level of abstraction for the system model. Part of the automotive dashboard analysis model is the cruise control statechart model and the gauge subassembly model, both of which are software models. The cruise control statechart model is an instance of a software model type called the statechart model. The cruise control statechart model is specified using the Rational Rose® implementation and is built by Joe Bloggs. The statechart model is implemented on the Rational Rose® and the With Class® implementations. The statechart model is based on two sources [7, 8].

Generation Reuse

Figure 3-11 illustrates the model that classifies historical data for reuse using generation. This model is from the Measurement Data Classification MLC at meta-level 1. A software model (the source model) is used to generate other software models (complete models). Generation of a complete model is done using some implementation. A source model contributes to a complete model by generating part or all of that model. The measure is generated by comparing the model generated from the source model (the generated model) to the complete model. A source model that is located on more than one implementation has a separate contribution measure for each implementation. One or more software developers generated the software model using some implementation. The implementation attached to the source model is responsible for the generated model.

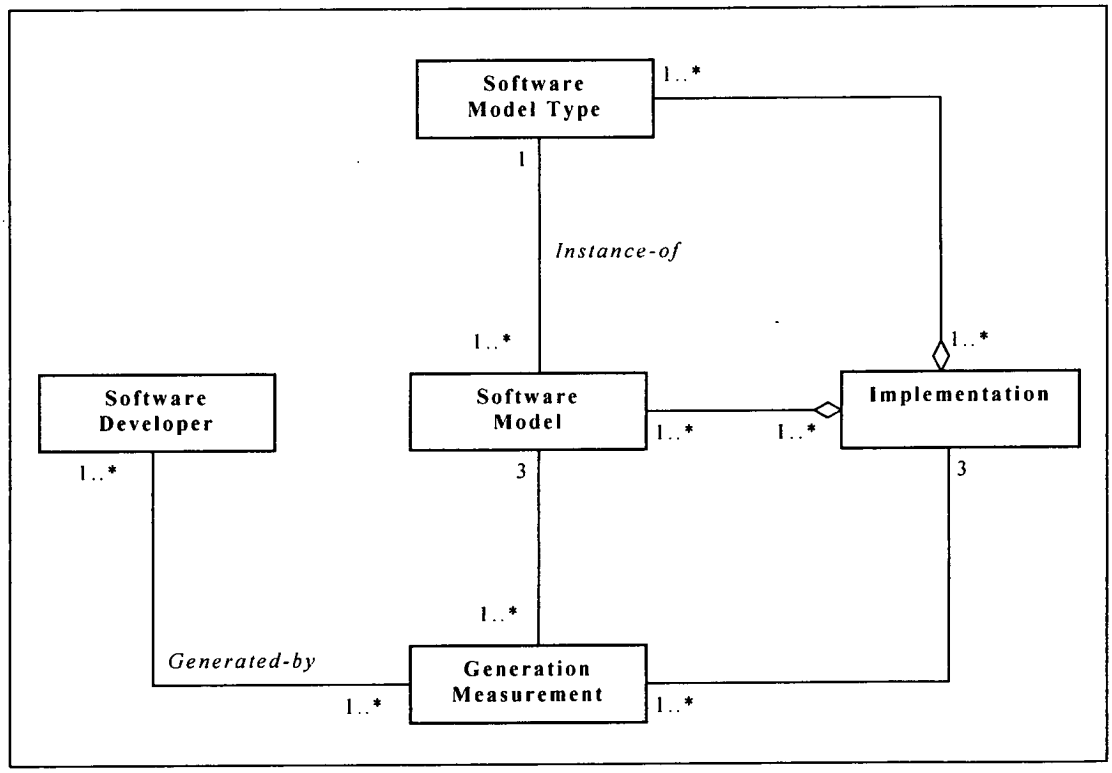

Figure 3-11: *Model for Data Classification of Generation Reuse*

A worked example: A source model called the RobotStateTransitionModel is an instance of the software model type called the StateTransitionModel. The StateTransitionModel is implemented on WithClass®, and the RobotStateTransitionModel is also located in it. The source model is used to generate the generated model called the RobotC++GeneratedModel. The generated model is used in the complete model called the RobotC++CompleteModel. Both the generated and complete model are instances of the software model type called the C++Model. The C++Model is implemented on Borland®C++, and both the

RobotC++GeneratedModel and RobotC++CompleteModel are located in it. The Generated model was generated by the software developer Joe Bloggs.

Composition Reuse

Figure 3-12 illustrates the model that classifies historical data for reuse using composition. This model is from the Measurement Data Classification MLC at meta-level 1. A software model (the library model) is reused in one or more software models (application models). A library model contributes to an application model when parts of the library model are also in the application model. The contribution is measured by comparing the library model with the application model. A library model that is located on more than one implementation makes a separate contribution for each implementation. Reuse of a library model is performed by one or more software developers.

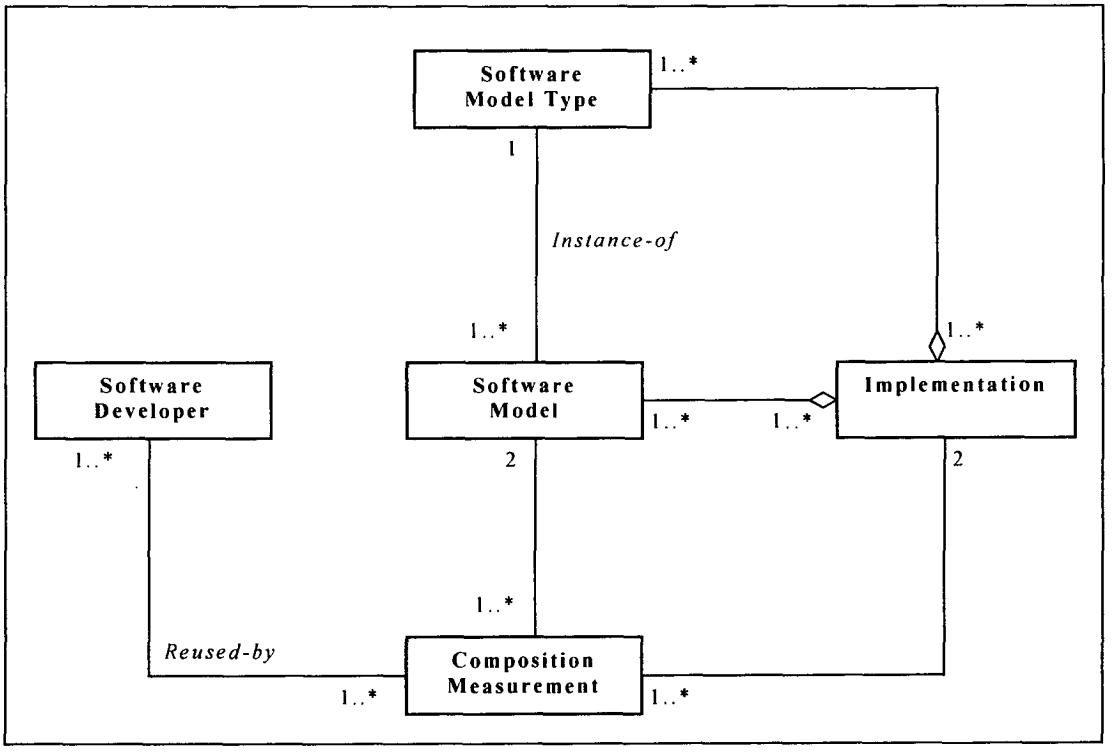

Figure 3-12: *Model for Data Classification of Composition Reuse*

A worked example: A library model, called the RobotLibraryModel is reused in an application model called the RobotApplicationModel. Both software models are instances of a software model type called the StateTransitionModel. The StateTransitionModel is implemented on WithClass®, and both the RobotLibraryModel and RobotApplicationModel are located in it. The RobotLibraryModel was reused in the RobotApplicationModel by the software developer, Joe Bloggs.

3.6 Set Theory

Fundamentals

Table 3-3 details the infinite sets used for measurement of reuse. To increase readability, a naming convention for sets was established and a notation is introduced to discriminate between sequences and sets.[1] The SoftwareModeTypes set characterises the structure of software model types. This set is from the Software Model Type Set Theory MLC at meta-level 2. A software model type contains a number of elements that name its model element types. The name of the set is the name of the software model type, e.g. the set ClassModel contains classes, operations, and parameters. This is defined as ClassModel = { Class, Operation, Parameter }.

Table 3-3: Fundamental Sets and Sequences with Examples

Set or Sequence	Informal Basis; Formal Rendition
Characters = {a..z,A..Z,0..9, }	N/A
String = { StringCharacter | StringCharacter ∈ Characters }^{sq}	Class Models have Classes; Class = { c, l, a, s, s }^{sq} Class ⊂ String
	My class is a class; MyClass = { M, y, C, l, a, s, s }^{sq} MyClass ⊂ String
CompoundIdentifiers = { IdentifierPart | IdentifierPart ⊂ String }^{sq}	My Class has a Set Name operation; MyClassSetName = { { M, y , C, l, a, s, s }^{sq}, { S, e, t, N, a, m, e }^{sq} }^{sq} MyClassSetName ⊂ CompoundIdentifiers
SoftwareModelTypes = { ModelElementType | ModelElementType ⊂ String }	A Class Model has Classes and Operations. Each operation has parameters; ClassModel = { Class, Operation, Parameter } ClassModel ⊂ SoftwareModelTypes Class = { C, l, a, s, s}^{sq} Operation = { O, p, e, r, a, t, i, o, n }^{sq} Parameter = { P, a, r, a, m, e, t, e, r }^{sq}
SoftwareModels = { (ModelElement, ElementIdentifier) | ModelElement ∈ SoftwareModelTypes ∧ ElementIdentifier ⊂ CompoundIdentifiers }	My Class Model has My Class. My Class has a Set Name operation; MyClassModel = { (Class, MyClass), (Operation, MyClassSetName) } MyClassModel ⊂ SoftwareModels

[1] Sets are named using a string of characters from the Latin alphabet. The first character in the set name must be in upper case, remaining characters can be in upper or lower case. For example, "Myset", "MYset", "MySet", "MysET" and "MYSeT" are valid set names; but "mYSET", "mySet", "mYseT", "myseT" and "myset" are not valid set names. A sequence is enclosed by braces with the right brace marked by a superscript "sq". For example, the sequence <a,s,e,q,u,e,n,c,e> or { (1,a), (2,s), (3,e), (4,q), (5,u) ,(6,e), (7,n), (8,c), (9,e) } is represented as { a, s, e, q, u, e, n, c, e, }^{sq}.

The SoftwareModels set characterises the structure of software models. These are instances of some software model type defined using the SoftwareModelTypes set. The SoftwareModels set is from the Software Model Set Theory MLC at meta-level 1. Each element in this kind of set represents the model elements it contains. The name of the set is the name of the software model, e.g. an instance of the ClassModel is MyClassModel = { (Class, MyClass), (Operation, MyClassSetName) }. Each element in the set is an ordered pair. The first part represents the name of a model element type, the second part represents the unique identifier for the model element. This unique identifier is referred to as the compound identifier, e.g. an Operation with the name SetName in a class called MyClass is defined as the element (Operation, MyClassSetName). Compound identifiers are derived from the composition hierarchy for software model types and software models. The structure of this hierarchy is described in section 3.7.

All software models are characterised as sets. These models include the generated model (GeneratedModel), source model (SourceModel) and complete model (CompleteModel) for generation reuse, and the library model (LibraryModel) and application model (ApplicationModel) for composition reuse. Each of these sets is a subset of the SoftwareModels set.

Table 3-4: Fundamental Functions and Examples

Function
$IsSoftwareModelType(A) = \begin{cases} \text{true if } A \subseteq SoftwareModelTypes \\ \text{false if } A \not\subset SoftwareModelTypes \end{cases}$ IsSoftwareModelType(ClassModel) = true IsSoftwareModelType(MyClassModel) = false
$IsSoftwareModel(B) = \begin{cases} \text{true if } B \subseteq SoftwareModels \\ \text{false if } B \not\subset SoftwareModels \end{cases}$ IsSoftwareModel(ClassModel) = false IsSoftwareModel(MyClassModel) = true
if $IsSoftwareModel(A) = true$ then $SoftwareModelType(A) = \{a : \forall a \in SoftwareModelTypes, \exists (a,b) \in A\}$ SoftwareModelType(MyClassModel) = { Class, Operation }

Generation Reuse

For generation reuse, the GeneratedModel and CompleteModel usually have the same set for their software model type. The size of each software model is calculated by counting the number of elements in each set. The amount generated is calculated by counting the number of elements in the GeneratedModel that are also in the CompleteModel. The waste generated is calculated by counting the number of elements in the GeneratedModel that are not in the CompleteModel. The amount not generated is calculated by counting the number of elements in the CompleteModel that are not in the GeneratedModel. The amount not generated reflects the additional work done to finish the complete model. Figure 3-13 illustrates measurement of reuse using generation with Venn diagrams.

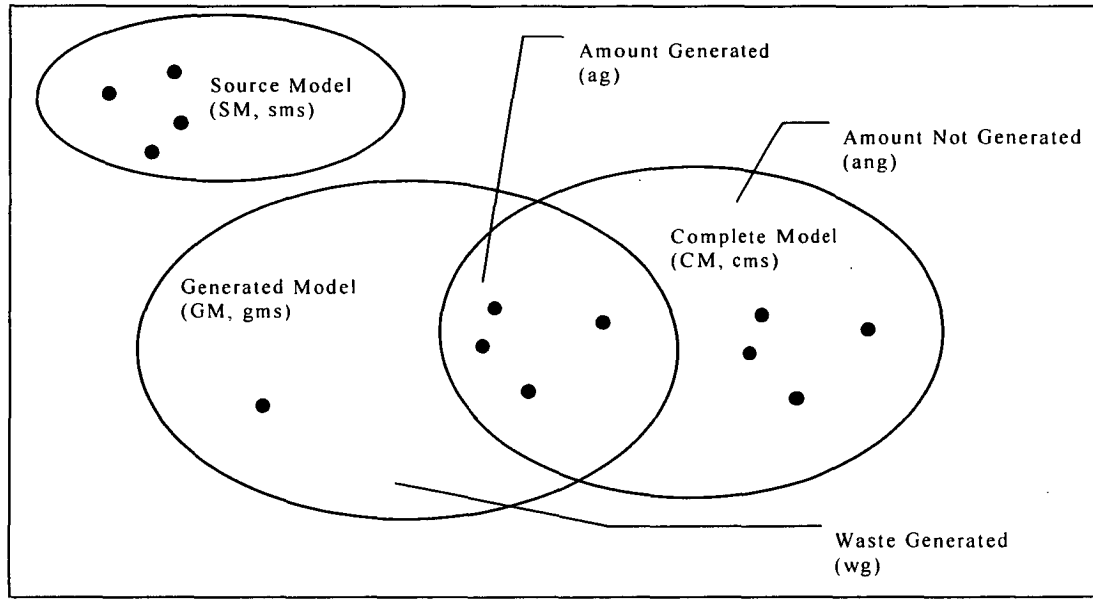

Figure 3-13: *Measuring Generation Reuse using Software models as Sets*

In applying set theory, the generated model, source model and complete model are all sets. They are named as GeneratedModel, SourceModel, and CompleteModel. Each of these sets is a subset of the SoftwareModels set. The GereatedModel set and CompleteModel set must have the same set for their software model type. The source model can be of any software model type. The size of each software model is equal to the cardinality of its set. The amount generated equal to the cardinality of the intersection of the GeneratedModel and the CompleteModel. The waste generated is equal to the complement of the CompleteModel relative to the GeneratedModel. The amount not generated is equal to the complement of the GeneratedModel relative to the CompleteModel. The values are measures of how much the SourceModel contributed to the complete model. Table 3-5 shows the use of set operators to gain the required numbers for the measure of generation reuse. Table 3-5 is based on the Measurement Data Classification MLC at meta-level 1.

Table 3-5: Calculations for Generation Reuse

Required Amount	Calculation
Size of Source Model (sms)	| SourceModel |
Size of Generated Model (gms)	| GeneratedModel |
Size of Complete Model (cms)	| CompleteModel |
Amount Generated (ag)	| GeneratedModel ∩ CompleteModel |
Amount Not Generated (ang)	| CompleteModel – GeneratedModel |
Waste Generated (wg)	| GeneratedModel – CompleteModel |
Percentage Contribution to Complete Model (pccm)	$\frac{ag}{cms} \times \frac{100}{1}$
Percantage Not Generated (png)	$\frac{ang}{cms} \times \frac{100}{1}$
Percentage of Waste Generation (pwg)	$\frac{wg}{gms} \times \frac{100}{1}$

Generation Reuse, an example: A source model called the RobotStateTransitionModel is an instance of the software model type called the StateTransitionModel. The StateTransitionModel is implemented on WithClass®, and the RobotStateTransitionModel is also located in it. The source model is used to generate the model called the RobotGeneratedModel. The generated model is used in the complete model called the RobotCompleteModel. Both the generated and complete model are instances of the software model type called the C++Model. The C++Model is implemented on Borland®C++, and both the RobotGeneratedModel and RobotCompleteModel are located in it.

1. The source model has four model elements (sms).
2. The generated model has five model elements (gms).
3. The complete model has eight model elements (cms).
4. Four of the model elements in the generated model are also in the complete model (ag).
5. Four of the model elements in the complete model are not in the generated model (ang).
6. One of the model elements in the generated model is not in the complete model (wg).
7. The percentage of the complete model that is generated is 50% (pccm).
8. The percentage of the complete model that is not generated is 50% (png).
9. The percentage that is generated but not used in the complete model is 20% (pwg).

A worked example using set theory: For the statechart model implemented on State Maker and Rational Rose a set is defined for its type;

StateTransitionModel = { State, Transition }
State = { S, t, a, t, e }
Transition = { T, r, a, n, s, i, t, i, o, n }

For a manufacturing robot, a software model is defined;

RobotStateTransitionModel = { (State, Idle), (State, Working), (Transition, Idle.Start), (Transition, Working.Stop) }

For the generated model and complete model there is also a software model type as follows;

C++Model = { Class, Operation, Type, Variable, Statement }

Then both the generated model and complete model will have separate software models defined as sets;

RobotGeneratedModel = { (Class, TRobot), (Operation, TRobot.Start), (Operation, TRobot.Stop), (Variable, TRobot.State), (Class, TRobotLink) }

RobotCompleteModel = { (Class, TRobot), (Operation, TRobot.Start), (Operation, TRobotStop), (Variable, TRobot.State), (Statement, TRobot.Start.1IfState=Idlethen), (Statement, TRobot.Start.2InitiateSeq()), (Statement, TRobot.Stop.1IfState=Workingthen), (Statement, TRobot.Stop.2HaltSeq()) }

For the amount generated, the intersection of the generated model and complete model is used.

RobotGeneratedModel ∩ RobotC++CompleteModel
= { (Class, TRobot), (Operation, TRobot.Start), (Operation, TRobot.Stop), (Variable, TRobot.State) }

For the amount not generated, the complete model minus the generated model is used.

RobotCompleteModel – RobotGeneratedModel
= { (Statement, TRobot.Start.1IfState=Idlethen), (Statement, TRobot.Start.2InitiateSeq()), (Statement, TRobot.Stop.1IfState=Workingthen), (Statement, TRobot.Stop.2HaltSeq()) }

For the waste generated, the generated model minus the complete model is used.

RobotGeneratedModel – RobotCompleteModel
= { (Class, TRobotLink) }

The amounts based on Table 3-5 are then calculated. This is shown in Table 3-6.

Table 3-6:Calculated Amounts for Generation Reuse

Required Amount	Calculation
Size of Source Model (sms)	| RobotStateTransitionModel | = 4
Size of Generated Model (gms)	| RobotGeneratedModel | = 5
Size of Complete Model (cms)	| RobotCompleteModel | = 8
Amount Generated (ag)	| RobotGeneratedModel ∩ RobotCompleteModel | = 4
Amount Not Generated (ang)	| RobotCompleteModel – RobotGeneratedModel | = 4
Waste Generated (wg)	| RobotGeneratedModel – RobotCompleteModel | = 1
Percentage Contribution to Complete Model (pccm)	$\frac{4}{8} \times \frac{100}{1} = 50\%$
Percentage Not Generated (png)	$\frac{4}{8} \times \frac{100}{1} = 50\%$
Percentage of Waste Generation (pwg)	$\frac{1}{5} \times \frac{100}{1} = 20\%$

Composition Reuse

For composition reuse, the LibraryModel and ApplicationModel usually have the same set for their software model type. The size of each software model is calculated by counting the number of elements in each set. The amount reused is calculated by counting the number of elements in the LibraryModel that are also in the ApplicationModel. The amount not reused is calculated by counting the number of elements in the LibraryModel that are not in the ApplicationModel. The amount added is calculated by counting the number of elements in the ApplicationModel that are not in the LibraryModel. Figure 3-14 illustrates measurement of reuse using composition with Venn diagrams.

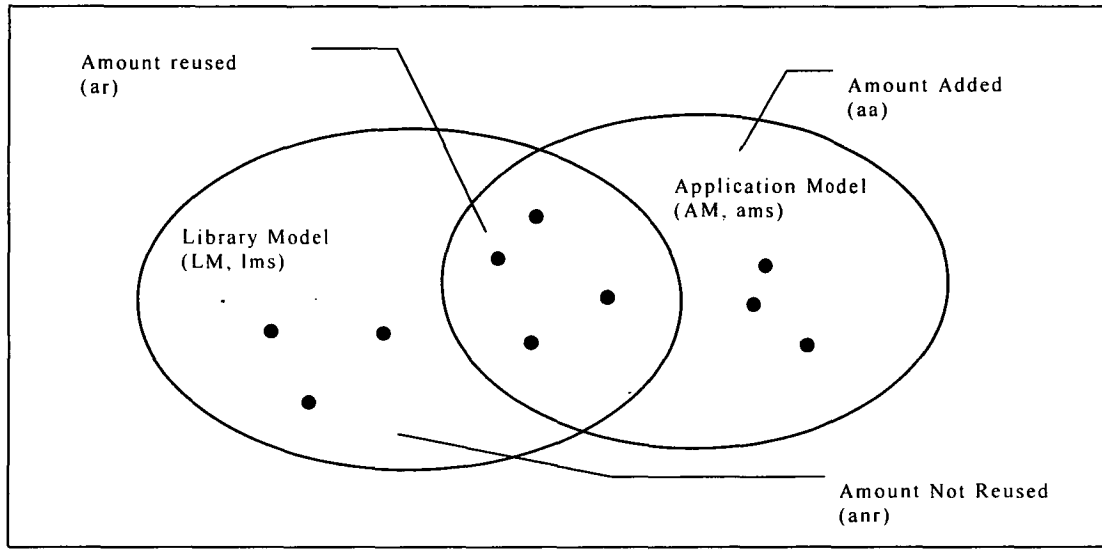

Figure 3-14: *Measuring Composition Reuse using Software models as Sets*

In applying set theory, the library model and application model are both sets. They are named as LibraryModel and ApplicationModel. Both sets are a subset of the SoftwareModels set. Both sets must have the same set for their software model type. The size of each software model is equal to the cardinality of its set. The amount reused is equal to the cardinality of the intersection of the LibraryModel and the ApplicationModel. The amount not reused is equal to the complement of the ApplicationModel relative to the LibraryModel. The amount added is equal to the complement of the LibraryModel relative to the ApplicationModel. The results are measures relevant to the LibraryModel. Table 3-7 shows the use of set operators to gain the required numbers for the measure of composition reuse. Table 3-7 is based on the Measurement Data Classification MLC at meta-level 1.

Table 3-7: Calculations for Composition Reuse

Required Amount	Calculation
Size of Library Model (lms)	| LibraryModel |
Size of Application Model (ams)	| ApplicationModel |
Amount Reused (ar)	| LibraryModel ∩ ApplicationModel |
Amount Added (aa)	| ApplicationModel – LibraryModel |
Amount Not Reused (anr)	| LibraryModel – ApplicationModel |
Percentage of Reuse in Application Model (pram)	$\frac{ar}{ams} \times \frac{100}{1}$
Percentage Added in Application Model (paam)	$\frac{aa}{ams} \times \frac{100}{1}$
Percentage of Library Model Not Reused (plmnr)	$\frac{anr}{lms} \times \frac{100}{1}$

Composition Reuse, an example: A library model, called the RobotLibraryModel is reused in an application model called the RobotApplicationModel. Both software models are instance of a software model type called the StateTransitionModel. The StateTransitionModel is implemented on WithClass®, and both the RobotLibraryModel and RobotApplicationModel are located in it.

1. The RobotLibraryModel contains seven model elements (lms).
2. The RobotApplicationModel contains seven model elements (ams).
3. Four of the model elements in the library model are reused in the application model (ar)
4. Three model elements in the application model are added (aa).
5. Three model elements in the library model are not reused (anr).
6. The percentage of the library model in the application model is 57% (pram).
7. The percentage of the application model that is added is 43% (paam).
8. The percentage of the library model that is not reused is 43% (plmnr).

A worked example using set theory: Firstly, the library model and application model must have a software model type.

StateTransitionModel = { State, Transition }
State = { S, t, a, t, e }
Transition = { T, r, a, n, s, i, t, i, o, n }

For the library model and application model, we define a software model as:

RobotLibraryModel = { (State, Idle), (State, Working), (Transition, Idle.Start), (Transition, Working.Stop), (State, Testing), (Transition, Idle.StartTest), (Transition, Testing.StopTest) }

RobotApplicationModel = { (State, Idle), (State, Working), (Transition, Idle.Start), (Transition, Working.Stop), (State, Maintenance), (Transition, Idle.MoveToMaint), (Transition, Maintenace.EndMaint) }

For the amount reused, the intersection of the library model and application model is used.

RobotLibraryModel ∩ RobotApplicationModel
= { (State, Idle), (State, Working), (Transition, Idle.Start), (Transition, Working.Stop) }

For the amount added, the application model minus the library model is used.

RobotApplicationModel – RobotLibraryModel
= { (State, Maintenance), (Transition, Idle.MoveToMaint), (Transition, Maintenance.EndMaint) }

For the amount not reused, the library model minus the application model is used.

RobotLibraryModel – RobotApplicationModel
= { (State, Testing), (Transition, Idle.StartTest), (Transition, Testing.StopTest) }

The amounts based on Table 3-7 are then calculated. This is shown in Table 3-8.

Table 3-8: Calculated Amounts for Composition Reuse

Required Amount	Calculation
Size of Library Model (lms)	| RobotLibraryModel | = 7
Size of Application Model (ams)	| RobotApplicationModel | = 7
Amount Reused (ar)	| RobotLibraryModel ∩ RobotApplicationModel | = 4
Amount Added (aa)	| RobotApplicationModel – RobotLibraryModel | = 3
Amount Not Reused (anr)	| RobotLibraryModel – RobotApplicationModel | = 3
Percentage of Reuse in Application Model (pram)	$\frac{4}{7} \times \frac{100}{1} = 57\%$
Percentage Added in Application Model (paam)	$\frac{3}{7} \times \frac{100}{1} = 43\%$
Percentage of Library Model Not Reused (plmnr)	$\frac{3}{7} \times \frac{100}{1} = 43\%$

3.7 Model Classification

Fundamentals

The model for classifying software model types and software models is illustrated in Figure 3-15. This model is from the Software Model Type Classification MLC at meta-level 2, and the Software Model Classification MLC at meta-level 1. The examples given are drawn from Figure 3-16. The elements in this model are maintained by the Model Classification component.

- A software model for developing software models is referred to as a *software model type*. A software model type has a name. For example, the Class Model underlying class diagrams is the name of a software model type. Software model types have instances referred to as software models. These are also named. For example, My Class Model is the name for an instance of the Class Model.
- Software model types contain elements referred to as *model element types*. These are also named. For example, the Class Model contains model element types called Classes and Associations. A model element type in a software model type has corresponding instances of it called *model elements*. For example, My Class Model contains elements such as "My Class" and "My Other Class". These are instances of Class. Model element types can also contain other model element types. For example, Classes contain model element types called Operations.
- Each model element type contains one or more *identifier part types* that uniquely identify it within the boundary of its aggregate model element type or software model type. For example, a Class has an identifier part type called "Class Name". This is unique within the Class Model. An Operation has an identifier part type called "Operation Name". This is unique within the Class.
- Each identifier part type has a number of corresponding *identifier parts* in a software model. For example, "My Class" has "MyClass" for a Class Name, and "My Other Class" has "MyOtherClass" for a Class Name. These are instances of Class Name. "My Class" contains an Operation called "Set Name" with "SetName" for an Operation Name. This is an instance of Operation Name.

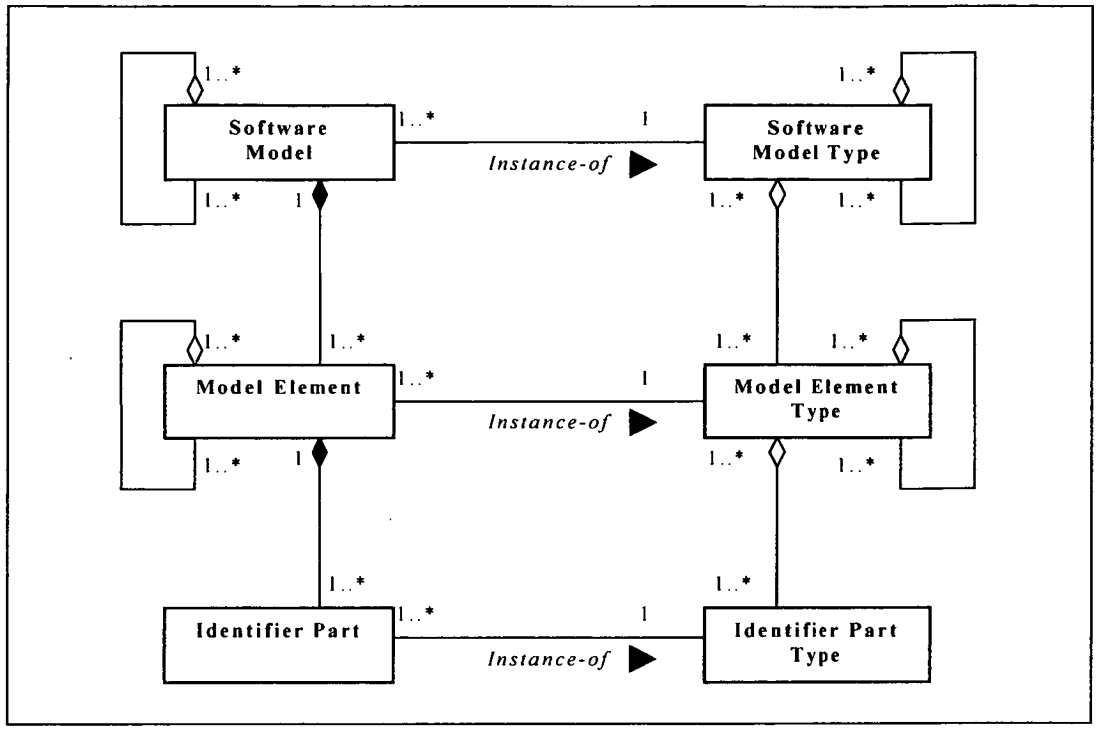

Figure 3-15: *Model for Classification of Software Models and Model Types*

Although a model element type usually has one identifier part type, there are cases where this is different. For example, a parameter in an operation is uniquely identified by its type and position (Two identifier parts). A compound identifier for a model element in a software model set is formed by concatenating the identifier parts for the model element itself with identifier parts from its aggregate model elements. For example, the Operation in the Class "My Class" has the compound identifier "MyClassMyOperation". Figure 3-16 illustrates this.

Models classified using the model in Figure 3-15 are represented using a text language. This is done for both software model types and software models. The structure of the text language for these representations is described using the text definition language (TDL). This is a variation of Extended Bakus-Naurr Form (EBNF). Refer to the following sources for EBNF [1, 5, 6].

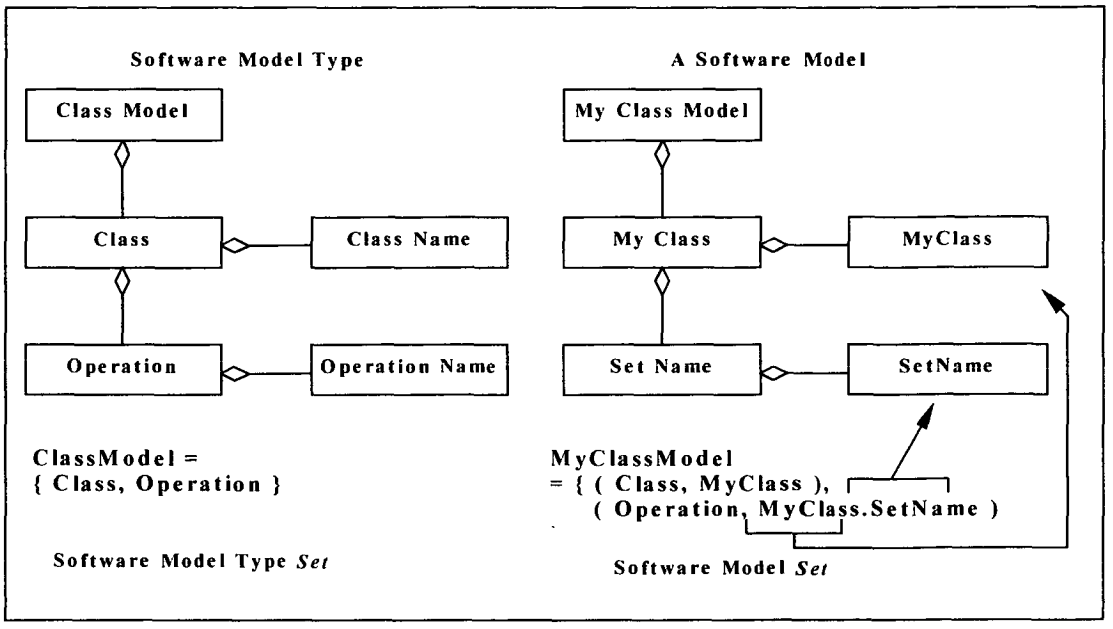

Figure 3-16: *Example of Model Classification of Software Models/ Model Types*

Representation of Software Model Types

Figure 3-17 shows the TDL specification for representing software model types. This specification is from the Software Model Type Classification MLC at meta-level 2. A software model type is represented using the <Software model type specification> non-terminal. Figure 3-18 is an example of the software model type for the C++ model used under the heading "Generation Reuse". Figure 3-19 is an example of the software model type for the state transition model used under the heading "Composition Reuse".

```
<Software model type specification> ::= <Software model type name>
<Model element types> <Software model type set>

<Software model type name> ::= Software Model Type Name: <Phrase>

<Model element types> ::= Model Element Types: '['
{ <Model element type> ; } ']'

<Model element type> ::= <Element type name> ; <Identifier part types> [
<Component element types> ]

<Element type name> ::= Element Type Name: <Phrase>

<Identifier part types> ::= Identifier part types: '[' { <Identifier part type> ; } ']'

<Identifier part type> ::= Part type name: <Phrase>

<Component element types> ::= Component element types: '['
{ <Model Element type> ; } ']'

<Software model type set> ::= Software Model Type Set: '{' <Phrase>
{ , <Phrase> } '}'
```

Figure 3-17: *Text representation for software model types*

The name of a software model type is represented using the <Software model type name> non-terminal. For example, a software model type is named as the "C++ Model" (Figure 3-18(a)), or the "State Transition Model" (Figure 3-19(a)).

Model element types associated with a software model type are represented using the <Model element types> non-terminal. Each model element type is identified using the <Model element type> non-terminal. For example, a C++ Model contains model element types named "Class" and "Type" (Figure 3-18(b)). A State Transition Model contains a model element type named "State" (Figure 3-19(b)).

Identifier part types that uniquely identify an instance of the element type are represented using the <Identifier part types> non-terminal in the <Model element type> non-terminal. Each identifier part is represented using the <Identifier part type> non-terminal contained in the <Identifier part types> non-terminal. For example, a Class in a C++ Model is uniquely identified using its name (Figure 3-18(c)). In a State Transition Model a Transition is uniquely identified using the source state of the transition and the associated event (Figure 3-19(c)).

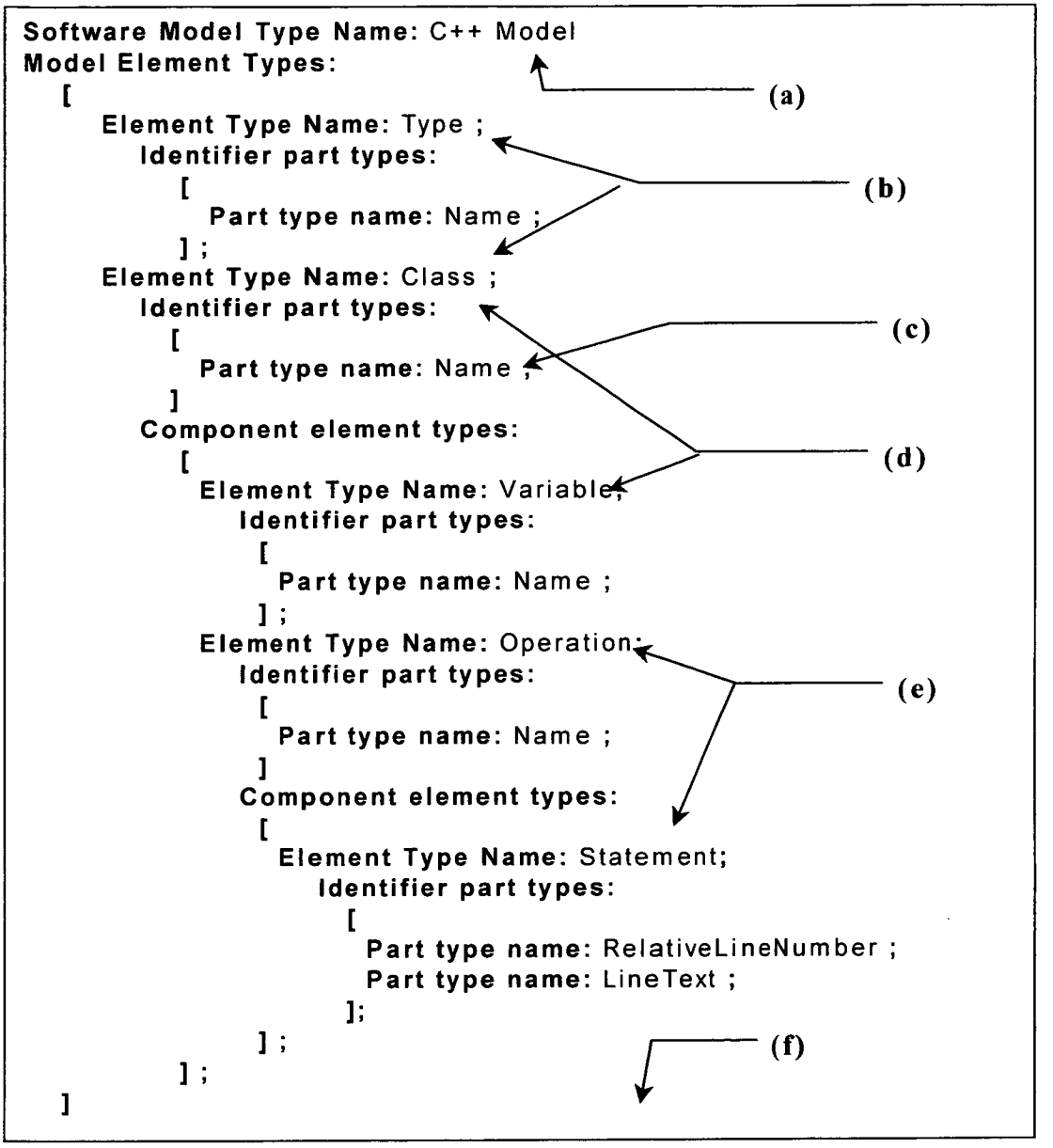

Figure 3-18: *Software Model Type Classification for C++ Model*

The <Component element types> non-terminal in the <Model element type> non-terminal is used if a model element type contains other model element types. For example, classes contain variables in a C++ Model (Figure 3-18(d)). Operations in Classes also contain statements (Figure 3-18(e)).

The set for a software model type is represented using the <Software model type set> non-terminal. Model element types contained by the software model type are named using the <Phrase> non-terminal. For example, a C++ Model contains classes, operations, types, variables, and statements (Figure 3-18(f)). The State Transition Model contains states and transitions (Figure 3-19(d)).

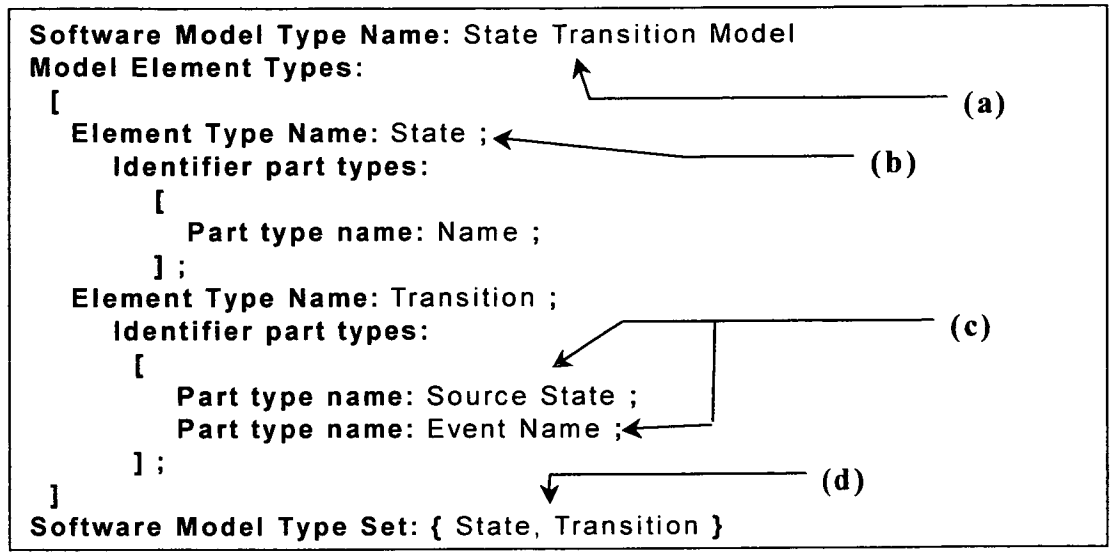

Figure 3-19: *Software Model Type Classification for State Transition Model*

Representation of Software Models

Figure 3-20 shows the TDL specification for representing software models. This specification is from the Software Model Classification MLC at meta-level 1. Software models are represented using the <Software model specification> non-terminal. Figure 3-21 is an example of a software model that is an instance of the software model type called the "C++ model" in Figure 3-18. This software model was represented as a set under the heading "Generation Reuse". Figure 3-22 is an example of a software model that is an instance of the software model type called the "State Transition Model" in Figure 3-19. This software model was represented as a set under the heading "Composition Reuse".

The name of a software model is represented using the <Software model name> non-terminal. For example, a software model is named the "Robot C++ Complete Model" (Figure 3-21(a)), or the "Robot Application Model" (Figure 3-22(a)).

```
<Software model specification> ::= <Software model name>
<Model elements> <Software model set>

<Software model name> ::= <Software model type name>
Software Model Name: <Phrase>;

<Software model type name> ::= Software Model Type Name: <Phrase>.

<Model elements> ::= Model Elements: '[' <Model element> ; ']'

<Model element> ::= <Element type name>. Element Name: <Phrase>;
<Identifier parts> [ <Component elements> ]

<Element type name> ::= Element Type Name: <Phrase>

<Identifier parts> ::= Identifier parts: '[' { <Identifier part> ; } ']'

<Identifier part> ::= <Identifier part type>. Part value: <Phrase>

<Identifier part type> ::= Part type name: <Phrase>

<Component elements> ::= Component elements: '[' { <Model Element> ; }
']'

<Software model set> ::= Software Model Set: '{'
[ <Software model element> {, <Software model element> } '}'

<Software model element> ::= '(' <Phrase>, <Compound identifier> ')'

<Compound identifier> ::= <Phrase> { '.' <Phrase> }
```

Figure 3-20: *Text representation for software models*

The name of a software model type for a software model is represented using the <Software model type name> non-terminal. For example, the Robot C++ Complete Model is an instance of the software model type called the "C++ Model" (Figure 3-21(b)). The Robot Application Model is an instance of the software model type called the "State Transition Model" (Figure 3-22(b)).

The software model type name for a software model must also appear as a software model type name for a software model type specification. For example, the software model type name for the Robot C++ Model is the name for the C++ Model (Figure 3-18(a)). The software model type name for the Robot Application Model is the name for the State Transition Model (Figure 3-19(a)).

Model elements associated with a software model are represented using the <Model elements> non-terminal. Each model element is specified using the <Model element> non-terminal. A model element has an element type and a name. For example, the Robot C++ Complete Model has a model element of type "Class" named as "Robot" (Figure 3-21(c)). The Robot Application Model has a model element of type "State" named as "Idle State" (Figure 3-22(c)).

The element type of a model element must have a corresponding element type in the software model type for the software model. For example, the "Robot" model element is of type "Class" (Figure 3-21(c)). This appears as an element type name in

the C++ Model (Figure 3-18(d)). The "Idle State" model element is of type "State" (Figure 3-22(c)). This appears as an element type name in the State Transition Model (Figure 3-19(b)).

The identifier parts that uniquely identify a model element are represented using the <Identifier parts> non-terminal in the <Model element> non-terminal. Each identifier part is specified using the <Identifier part> non-terminal. Each identifier part has a part type and a value. For example, the "Robot" model element has an identifier part of type "Name" with value "TRobot" (Figure 3-21(d)). The model element of type "Transition" and named "Initialising Transition" has two identifier parts. The first is of part type "Source State" and has the value "Idle". The second is of part type "Event" and has the value "Start" (Figure 3-22(d)).

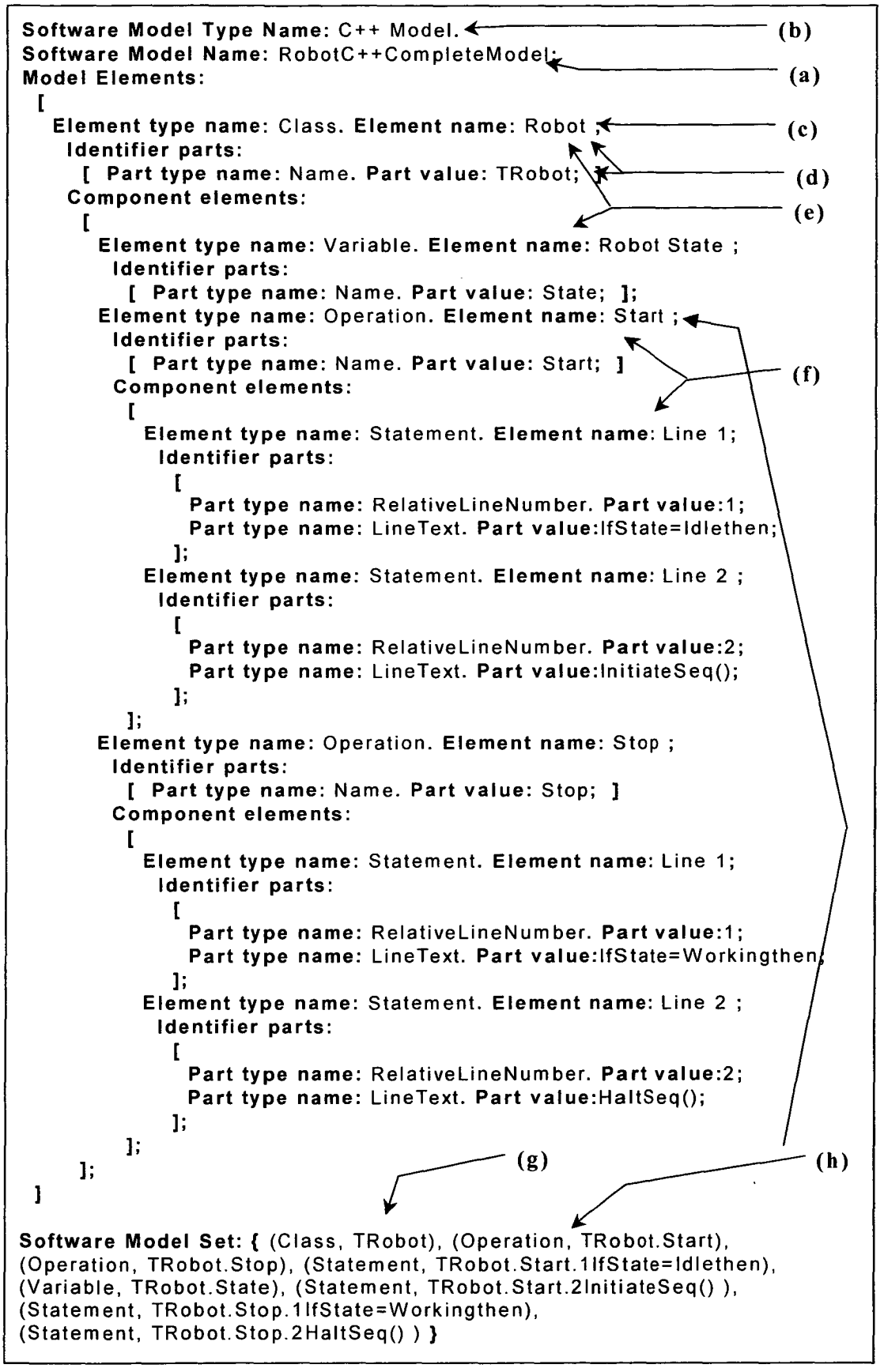

Figure 3-21: *Example text representation of a C++ Model Instance*

The part type name for a model element must have a part type name for its corresponding model element type in the software model type for the software model. For example, the "Robot" model element is of element type "Class" and has an identifier part of part type "Name" (Figure 3-21(d)). This part type name also appears as a part type name for the model element type named "Class" in the C++ Model (Figure 3-18(c)). The "Initialising Transition" model element is of element type "Transition" and has two identifier parts, one of part type "Source State" and the other of part type "Event" (Figure 3-22(d)). These names also appear as part type names in the model element type named "Transition" in the State Transition Model (Figure 3-19(c)).

The <Component elements> non-terminal in the <Model element> non-terminal is used when a model element contains other model elements. For example, the "Robot" model element contains the model element of type "Variable" and named "Robot State" (Figure 3-21(e)). The model element of element type "Operation" and named "Start" contains the model element of element type "Statement" and named "Line 1" (Figure 3-21(f)).

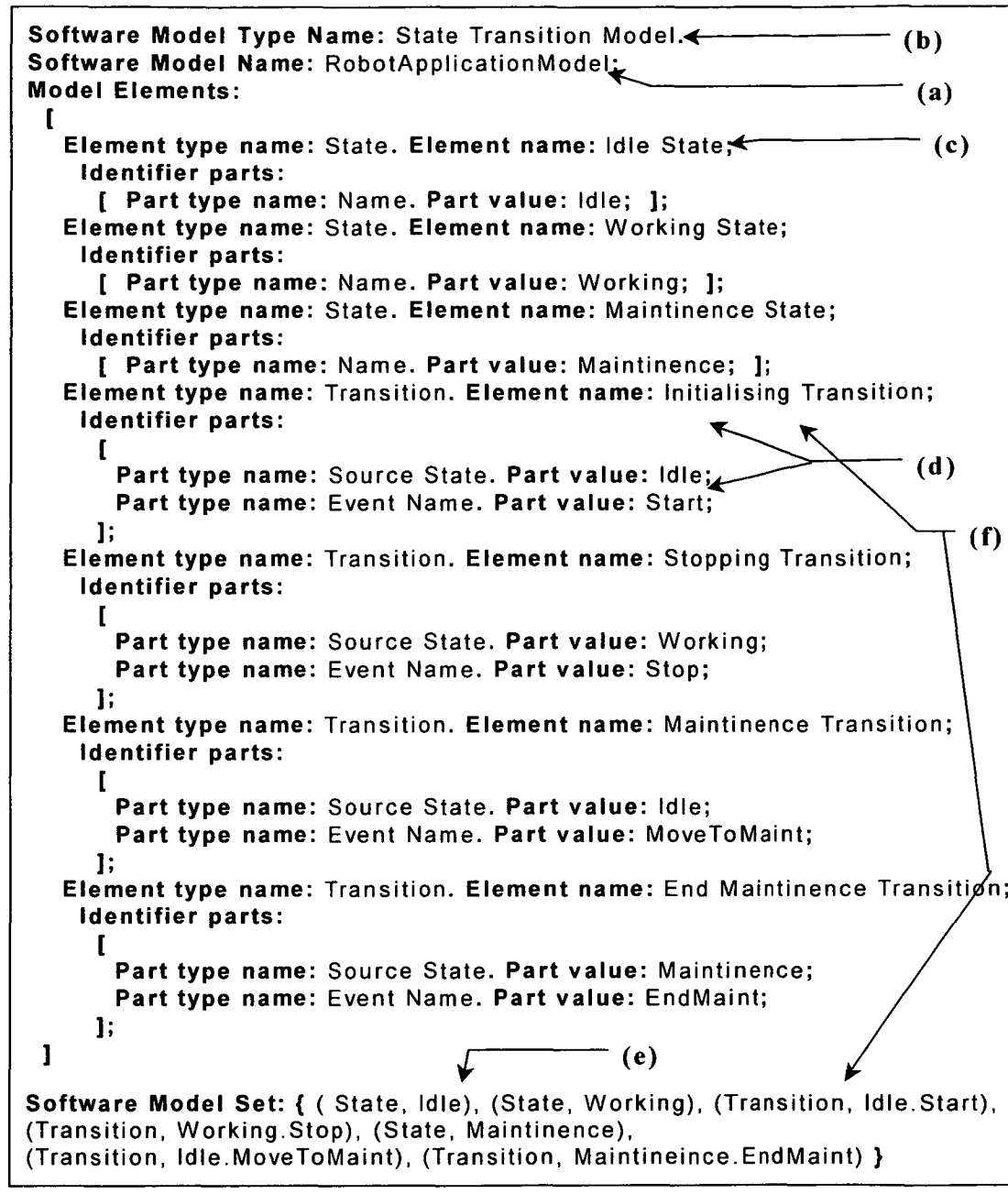

Figure 3-22: *Example Text Representation of a State Transition Model Instance*

Composition of the model elements in a software model must correspond to composition of the model element types in the software model type of the software model. For example, the "Robot" model element of element type "Class" contains the "State" model element of element type "Variable" (Figure 3-21(e)). The model element type named "Class" also contains the model element type named "Variable" in the C++ Model (Figure 3-18(d)). The "Start" model element of element type "Operation" contains the "Line 1" model element of element type "Statement" (Figure 3-21(f)). The model element type named "Operation" also contains the model element type named "Statement" in the C++ Model (Figure 3-18(e)).

The set for a software model is represented using the <Software model set> non-terminal. Model elements contained in the software model are represented using the <Software model element> non-terminal. The element type (<Phrase> in <Software model element>) and the unique identifier (<Compound identifier> in <Software model element>) are included for each model element. For example, the Robot C++ Complete Model contains one model element of element type Class ("TRobot"), two model elements of element type Operation ("Start", "Stop"), one model element of element type Variable ("State"), and four model elements of element type Statement ("1IfState=Idlethen", "2InitiateSeq()", "1IfState=Workingthen", "2HaltSeq()") (Figure 3-21(g)). The Robot Application Model contains three model elements of element type State ("Idle", "Working", "Maintenance") and four model elements of element type "Transition" ("Idle.Start", "Working.Stop", "Idle.MoveToMaint", "Maintinence.EndMaint") (Figure 3-22(e)).

The element type name for each model element in the software model set must also exist as a type name in the <Model elements> non-terminal for the software model specification. For example, the software model element of element type "Operation" and identified as "TRobotStart" in the set also appears as a model element of type "Operation" named "Start" in the Robot C++ Complete Model (Figure 3-21(h)) The software model element of element type "Transition" and identified as "IdleStart" in the software model set also appears as a model element of element type "Transition" named "Initialising Transition" in the Robot Application Model (Figure 3-22(f)).

Static and Dynamic Part Revisited

It can be seen from the above that there is a static part and dynamic part to the meta-model architecture. More specifically, Figure 3-15 was derived from the static part of the meta-model architecture for classification of software models and software model types. Figure 3-16 gives an example of the dynamic part of the meta-model architecture derived from the static part, i.e. from Figure 3-15.

Representation of software models and software model types follow a similar pattern. Figure 3-17 was derived from the static part of the meta-model architecture for representation of software model types. Figure 3-18 and Figure 3-19 are examples of the dynamic part of the meta-model architecture derived from the static part (Figure 3-17). Figure 3-20 was derived from the static part of the meta-model architecture for software models. Figure 3-21 and Figure 3-22 are examples of the dynamic part of the meta-model architecture derived from the static part (Figure 3-20).

3.8 Automation Specification

Figure 3-23 illustrates the components in the tool for measurement of reuse. Note that this thesis is concerned primarily with the Measurement Generator and Model Classification components for formulation of measures for the amount of reuse. The remaining components are mentioned for completeness and are not a major part of this thesis. All components were implemented using Borland C++ version 5.02 with Database Tools.

The *Context Data* component implements data classification from section 3.5. The context data component supports editing and modifying of qualitative data related to measurement of reuse. This includes variables such as developers, implementations, software projects, system models, and the relationships between these variables.

The *Model Classification* component implements model classification from section 3.7. The model classification component supports classification of software models and software model types measure of reuse for a range of software model types. Users classify software model types and then enter data or import data for software models of a given type.

The *Measurement Generator* component implements set theory from section 3.6. The measurement generator component uses the software models and software model types to measure reuse based on generation and composition. Users select software models and implementations that participate in an instance of reuse. An instance of measurement for reuse is then calculated with appropriate values using an automated version of set theory by transforming the software models classified by the model classification component into sets and using set operators to calculate the amount of reuse as defined in section 3.6.

The *Data Management* component supports importing and exporting of software models based on their software model type, and management of files for storage of data.

The *Test Analyser* component generates and executes a series of test cases for composition and generation based reuse. These are partly random, and based on a series of parameters supplied by the user.

The *Data Analysis* component relates measurement data to context data to support comparison and decision making.

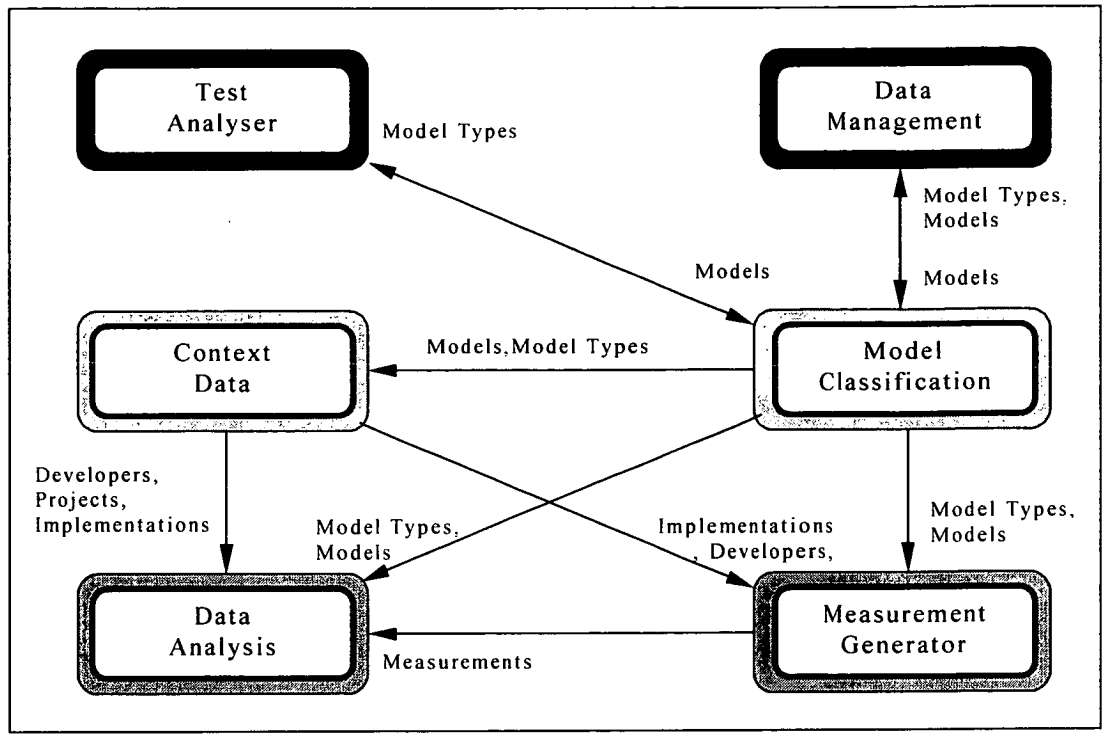

Figure 3-23: *Overview of Automation Specification*

The Context Data Component

The context data component is responsible for implementing data classification from section 3.5. For each class and most relationships in Figure 3-10, Figure 3-11, and Figure 3-12, a function was implemented as separate menu item and dialog box to provide updating of data related to data classification. The functions are named and described in Table 3-9, along with the database tables that are updated by the functions.

Table 3-9: Functionality of Context Data Component

Menu Item	Function	Database Table Updated
General | Software Developer	Provides addition, modification, and deletion of software developers	Software Developer.db
General | Software Project	Provides addition, modification, and deletion of software projects	Software Projects.db
General | System Model	Provides addition, modification, and deletion of system models	System Model.db
General | Implementation	Provides addition, modification, and deletion of Implementations	Implementation.db
General | Literature Source	Provides addition, modification, and deletion of Literature Sources	Literature Source.db
General | Link | Literature Sources to Software Model Type	Links a literature source to a software model type	Software Model Type BO Literature Source.db
General | Link | Software Model Type to Implementation	Links a software model type to an implementation	Software Model Type IO Implementation
General | Link | Software Model to Implementation	Links a software model to an implementation	Software Model LI Implementation.db
General | Link | System Model to Software Model	Links a system model to a software model	System Model CO Software Model.db
General | Link | Software Developer to Software Model	Links a software developer to a software model	Software Model MB Software Developer.db
General | Link | Software Developer to Model Element	Links a software developer to a model element in a software model	Software Developer HM Model Element.db
General | Link | Software Developer to Generation Reuse	Links a software developer to an instance of generation reuse	Software Model GB Software Developer.db
General | Link | Software Developer to Composition Reuse	Links a software developer to an instance of composition reuse	Software Model RB Software Developer.db

Composition of software model types, composition of software models, and composition of software models and model elements is maintained by the model classification component. The generation measurement class and associated relationships in Figure 3-11, and the composition measurement class and associated relationships in Figure 3-12 are maintained by the measurement generator component.

The Model Classification Component

The model classification component is responsible for implementing the classification of software model types and software models based on section 3.7. For each class and each relationship in Figure 3-15, a function was implemented as separate menu item and dialog box to provide updating of data related to model classification. The functions are named and described in Table 3-10, along with the database tables that are updated by the functions.

Table 3-10: Functionality of Model Classification Component

Menu Item	Function	Database Table Updated
Classify | Types | Software Model Type	Provides addition, modification, and deletion of software model types	Software Model Type.db
Classify | Types | Model Element Type	Provides addition, modification, and deletion of model element types	Model Element Type.db
Classify | Types | Identifier Part Type	Provides addition, modification, and deletion of identifier part types	Identifier Part Type.db
Classify | Relationships | Software Model Type -> Software Model Type	Implements composition between software model types.	Software Model Type CO Software Model Type.db
Classify | Relationships | Software Model Type -> Model Element Type	Implements composition between Software model types and model element types.	Software Model Type CO Model Element Type.db
Classify | Relationships | Model Element Type -> Model Element Type	Implements composition between model element types.	Model Element Type CO Model Element Type.db
Classify | Relationships | Model Element Type -> Identifier Part Type	Implements composition between model element types and identifier part types.	Model Element Type CO Identifier Part Type.db
Classify | Instances | Software Model	Provides addition, modification, and deletion of software models based on their software model type.	Software Model.db
Classify | Instances | Model Element	Provides addition, modification, and deletion of model elements based on their model element type.	Model Element.db
Classify | Instances | Identifier Part	Provides addition, modification, and deletion of identifier parts for model elements based on their identifier part type.	Identifier Part.db
Classify | Links | Software Model -> Software Model	Links a software model to another software model.	Software Model CO Software Model.db
Classify | Links | Software Model -> Model Element	Links a software model to a model element.	Software Model CO Model Element.db
Classify | Links | Model Element -> Model Element	Links a model element to another model element.	Model Element CO Model Element.db

A software model is classified after a software model type is classified. Measurement of the amount of reuse is done after software models are classified.

The Measurement Generator Component

The measurement generator component is responsible for implementing the set theory in section 3.6 and maintaining the results of measures for the generation measurement and composition measurement. For the composition measurement and generation measurement classes and their associations with software models in Figure 3-11 and Figure 3-12, a function was implemented as a separate menu item and dialog box to provide updating of data related to measurement of the amount of reuse. The functions are named and described in Table 3-11, along with the database tables that are updated by the functions.

Table 3-11: Functionality of Measurement Generator Component

Menu Item	Function	Database Table Updated; Classes Used
Measure | Generation Reuse	Measures generation reuse for software models.	Software Model GF Software Model.db; TMeasurement::MeasureGenerationReuse
Measure | Composition Reuse	Measures composition reuse for software models.	Software Model RI Software Model.db; TMeasurement::MeasureCompositionReuse

A class responsible for transforming the software models into sets was implemented and named the TMeasurement class. This class is also used to by the functions for the measurement generator component (See Table 3-11). There are two main methods for this class measure the amount of reuse using composition and generation. They are:

1. TMeasuremet::MeasureGenerationReuse is used to measure the amount of reus using generation. This method has the following signature:

   GeneratedModelType: Identifies the software model type of the generated model (Input). For example, using the example in section 3.6 for generation reuse this would equal C++ Model.

   GeneratedModelName: Identifies the software model that is the generated model (Input). For example, using the example in section 3.6 for generation reuse this would equal RobotGeneratedModel.

   CompleteModelType: Identifies the software model type of the complete model (Input). For example, using the example in section 3.6 for generation reuse this would equal C++ Model.

   CompleteModelname: Identifies the software model that is the complete model (Input). For example, using the example in section 3.6 for generation reuse this would equal RobotCompleteModel.

AmountGenerated: The amount generated as defined in section 3.6 for generation reuse (output).

AmountNotGenerated: The amount not generated as defined in section 3.6 for generation reuse (output).

WasteGenerated: The waste generated as defined in section 3.6 for generation reuse (output).

PctContributionToCompleteModel: The percentage contribution to complete model as defined in section 3.6 for generation reuse (output).

PctNotGenerated: The percentage not generated as defined in section 3.6 for generation reuse (output).

PctWasteGenerated: The percentage of waste generated as defined in section 3.6 for generation reuse (output).

2 TMeasurement::MeasureCompositonReuse is used to measure the amount of reuse using composition. This method has the following signature:

LibraryModelType: Identifies the software model type of the library model (Input). For example, using the example in section 3.6 for composition reuse this would equal StateTrasitionModel.

LibraryModelName: Identifies the software model that is the library model (Input). For example, using the example in section 3.6 for composition reuse this would equal the RobotLibraryModel.

ApplicationModelType: Identifies the software model type of the application model (Input). For example, using the example in section 3.6 for composition reuse this would equal StateTrasitionModel.

ApplicationModelName: Identifies the software model that is the application model (Input). For example, using the example in section 3.6 for composition reuse this would equal RobotApplicationModel.

AmountReused: The amount reused as defined in section 3.6 for composition reuse (output).

AmountAdded: The amount added as defined in section 3.6 for composition reuse (output).

AmountNotReused: The amount not reused as defined in section 3.6 for composition reuse (output).

PctContributionToApplicationModel: The percentage of reuse in application model as defined in section 3.6 for composition reuse.

PctAddedToApplicationModel: The percentage added in application model as defined in section 3.6 for composition reuse.

PctLibraryModelNotReused: The percentage of library model not reused as defined in section 3.6 for composition reuse.

To generate the measures for reuse, it is necessary to find a way to transform the software models stored in database tables into sets. This is done using three things:

1. A class that implements the classification of software models as sets (TSoftwareModelSet).
2. A class that implements a set of SQL queries to extract the software models for measurement (TApplicationQuery).
3. Additional methods in the TMeasurement class to perform the transformation.

Documentation of Specification

The automation specification is documented in three sections:

1. **Component Specification.** Logically related system functions are grouped into components. System functions are manifest in menu items.
2. **Automation Element Specification.** A series of tables are used to classify the automation specification using implementation specific terms.
3. **Automation Specification Mapping.** A series of tables that map MLCs in the measurement framework to automation elements in the automation specification.

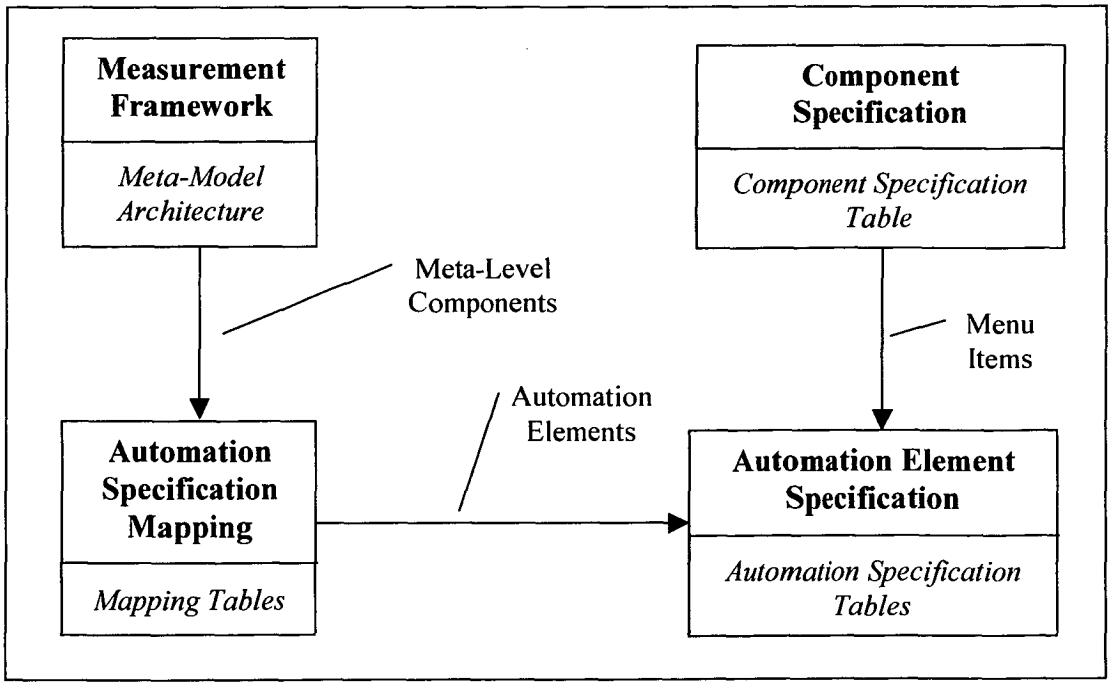

Figure 3-24: *Components of automation specification.*

Figure 3-24 illustrates the relationship between the automation specification and the measurement framework. The automation specification is used to justify conclusions made about the measurement framework based on its manifestation in a working prototype by using this prototype in the assessment framework.

Component Specification

Each component illustrated in Figure 3-23 is classified using the *component specification table*. Figure 3-25 illustrates an example of a component specification table. In this example, the name of the component is Model Classification. Three menu items form part of the component. These are Classify | Types | Software Model Type, Classify | Types | Model Element Type, and Classify | Types | Identifier Part Type. Further details about component specification tables can be found in Appendix B1.

Component **Specification Table**

Component Name: Model Classification

Menu Items
Classify | Types | Software Model Type
Classify | Types | Model Element Type
Classify | Types | Identifier Part Type

Figure 3-25: *Example of a component specification table*

Automation Element Specification

Classification of the automation specification is also done using a number of tables referred to as *automation specification tables*. There is one table for each kind of implementation specific element. An implementation specific element is referred to as an automation element. The kinds of automation elements classified were menu items, database tables, dialog boxes, templates, classes, structs, and functions. Further information related to the automation elements can be found in the relevant technical literature on the software used to develop the prototype tool [9-13].

Resource ID	Menu Entry	Class	Class Method
Classify	Classify |	[POPUP Only]	N/A
CM_SOFTWAREMODELTYPE	Types | Software Model Type	TCaseMfkWindow	CmSoftwareModelType
CM_MODELELEMENTTYPE	Types | Model Element Type	TCaseMfkWindow	CmModelElementType
CM_IDENTIFIERPARTTYPE	Types | Identifier Part Type	TCaseMfkWindow	CmIdentifierPartType

Figure 3-26: *Example of automation specification table for menu items.*

Figure 3-26 is an example of an automation specification table for menu items. Four menu items are defined. These are Classify | Types | Software Model Type Types, Classify |Types | Model Element Type, and Classify |Types | Identifier Part Type. The menu

item Classify | does not perform a function and groups the items below it as sub-menu items. Each menu item is related to a resource id (menu item name), a class, and a command. For example, the Classify | Types | Software Model Type is related to the resource id `CM_SOFTWAREMODELTYPE`, the class `TCaseMfkWindow`, and the command `CmSoftwareModelType`. Further details about automation element specification tables can be found in Appendix B2.

Automation Specification Mapping

For each kind of automation element there is a mapping from the MLCs in the measurement framework to automation elements in the automation specification. This is done using *mapping tables*. There was a separate mapping table for menu items, database tables, dialog boxes, templates, classes, structs, and functions.

MLC to Menu Item **Mapping Table**

Meta-Model Architecture			Automation Specification
MLC Name	**Definition Method**	**Definition Entry Part**	**Menu Item Name**
Software Model Type Classification	UML	**class** Software Model Type	CM_SOFTWAREMODELTYPE
Software Model Type Classification	UML	**class** Model Element Type	CM_MODELELEMENTTYPE
Software Model Type Classification	UML	**class** Identifier Part Type	CM_IDENTIFIERPARTTYPE

Figure 3-27: *Example of a mapping table for menu items*

Figure 3-27 is an example of a menu item mapping table. In this example the software model type classification is mapped to three menu items. For example the "**class** Software Model Type" that is part of the Software Model Type Classification MLC is mapped to the menu item CM_SOFTWAREMODELTYPE. Further details about mapping tables can be found in Appendix B3.

3.9 Summary

Now that the measurement framework is described there needs to be some way to test if the measurement framework can measure the amount of reuse for different kinds of software models. This is the subject of chapter 4.

Chapter 4 Assessment Framework

Chapter 3 described a "measurement framework" based on meta-modelling and asserted that it could be used for measuring the amount of reuse for different kinds of software models. This chapter is concerned with defining experiments that:

- Evaluate the measurement framework to see if it can measure the amount of reuse for different kinds of software models.
- Test the hypotheses using the measurement framework.

The next section gives an overview of chapter 4 to provide an introduction of how the experiments fulfil both of the above points.

4.1 Chapter Overview

This chapter has six main sections. These are:

4.2 ***Thesis Methodology*** illustrates how the experiments in chapter 4 are related to the rest of this thesis. The issues identified in Chapter 2 that are addressed by the measurement framework in chapter 3 are identified, along with how the experiments in chapter 4 are designed to test the hypothesis using the measurement framework. The experiments in chapter 4 provide results for discussion in chapter 5 and issues for further research in chapter 6.

Chapter 2 showed that measurement of the amount of reuse was inconsistent and incomplete for analysis and design models, including UML. The experiments, referred to as methodology phases, test the hypothesis using the measurement framework because each experiment must:

- Identify the hypothesis tested. This sets a goal for the experiment relevant to this thesis.
- Identify the meta-level components evaluated from the measurement framework. This ensures that the measurement framework is used in the experiment.
- Define hypothesis measures that indicate the level of support for a hypothesis. This defines an indicator to see if the experiment fulfilled the goal defined by the hypothesis.

4.3 ***Criteria for Success*** shows how the measurement framework in chapter 3 requires four experiments to verify that the measurement framework:

- Can measure the amount of reuse for different kinds of software models.

- The same components are used to measure the amount of reuse for different kinds of software models.

The four experiments are called software model type classification, software model classification, measurement testing, and automation assessment. The first two experiments are designed to test the prerequisites defined by the measurement framework for measuring the amount of reuse. The third experiment tests the measurement framework to see if it can measure the amount of reuse for different kinds of software models. The fourth experiment tests that the framework is general by identifying the common components used to perform the first three experiments. Since the measurement framework claims to measure the amount of reuse for different kinds of software models, the experiments can also test the hypotheses because the hypotheses assert that a measurement framework based on meta-modelling can measure the amount of reuse for different kinds of software models.

4.4 ***Hypotheses and Sub-Hypotheses*** revisits the hypothesis and describes a number of sub-hypothesis to break down the problem of analysis in the experiments into a manageable suitable for defining hypothesis measures, and align testing of the hypotheses with the experiments by having each experiment name the sub-hypotheses tested by it.

4.5 ***Dimensions of the Assessment Framework*** describes the essential aspects of a framework to classify the experiments (the assessment framework). The assessment framework has three dimensions:

1. ***Methodology Phases:*** This dimension represents the process used to conduct the experiments mentioned in section 4.3. There are four methodology phases, one for each experiment listed in section 4.3. Hence, the methodology phases are software model type classification, software model classification, measurement testing, and automation assessment.
2. ***Reuse Approach:*** This dimension partitions each experiment by reuse approach. Namely, internal composition, external composition, internal generation, and external generation. This is done to break down collection and analysis of data into more manageable problems.
3. ***Software Model Types and Implementations:*** This dimension represents the software model types implemented on CASE tools that are used in the experiments. More data is gathered as more and more software model types are used in the experiments. When more data is gathered this dimension expands whereas the previous two dimensions remain the same.

4.6 ***Methodology Phase Specification*** describes a template used to specify the methodology phases (the methodology phase description template). The template has three main sections:

- The ***Collection Process*** specifies how to conduct the experiment.
- ***Variables*** specify the independent and dependent variables in the experiment to evaluate the measurement framework against the claims made in section 4.3
- The ***Analysis Process*** specifies how to analyse data from the collection process to determine support for hypotheses via sub-hypotheses in section 4.4.

The template was defined and used for the following reasons: to ensure the experiments are specified more systematically, to ensure that the experiments test hypotheses in section 4.4 use the measurement framework, and to ensure that the measurement framework is evaluated against the claims made in section 4.3.

4.7 ***Methodology Phase Descriptions*** gives a description of each experiment based on the methodology phase description template. There are four methodology phases. These are:

1. ***Software Model Type Classification*** is an experiment designed to see if the measurement framework can classify different kinds of software models.
2. ***Software Model Classification*** is an experiment designed to see of the measurement framework can classify different software models and measure the size of these models using the software model type classifications from the software model type classification methodology phase.
3. ***Measurement Testing*** is an experiment designed to see if the measurement framework can measure the amount of reuse for different software models using the software model type classifications from the from the software model type classification methodology phase.
4. ***Automation Assessment*** is an experiment designed to see if the framework can measure the amount of reuse without relying on meta-level components in the framework unique to a particular software model type, unique implementation components in an automated version of the framework, or numerous conditional tests in the source code to account for different software model types or different software models.

It is useful to think of the first three methodology phases as a process for system testing of software that implements the measurement framework. Test cases are

developed to see if the measurement framework can classify different software model types, classify different software models, measure the size of different software models, and measure the amount of reuse for different software models. Expected and actual values are required for each test and this is what is defined as part of the collection process in the first three experiments. For example, a test case to see if the measurement framework measures composition reuse between a class model of size 0 as the library model and a class model of size 100 as the application model. The expected value for amount reused is 0.

The last methodology phase can be viewed as a trace of program execution. Use of the automated version of the measurement framework is logged to detect which implementation components are used to perform a given function in a test. For example, what meta-level components are used for classification of a class model with 100 model elements, or measurement of composition reuse between an empty class model and a non-empty class model of size 100 model elements.

The next section shows how the assessment framework is related to the rest of this thesis.

4.2 Thesis Methodology (Synopsis of the Argument)

An expanded roadmap of this thesis is illustrated in Figure 4-1. The following is a brief summary that shows:

- How the experiments (the assessment framework) are used to test the measurement framework against the hypotheses.
- How the experiments are related to the rest of this thesis.

Chapter 2 shows that measurement of the amount of reuse had the following limitations:

- There are a variety of measures for the amount of reuse.
- Each of them is only applied to a specific software model type, or small set of software model types.
- Measurement of the amount of reuse for object-oriented and UML models are not well addressed.
- Little consideration is given to the interpretation of the measures for assessment and improvement of practice.

This thesis addressed these basic limitations by:

- Defining a problem statement in Chapter 1 that includes:
  - Assessment of limitations of the measures for the amount of reuse (**RQ-2**, **H2**), and
  - Measurement of a range of software model types, including object-oriented and UML software models.
- Proposing a series of measures for the amount of reuse that includes how they are interpreted (Chapter 3 - the *measurement framework*).
- Defining the measures for the amount of reuse in terms of *meta-level components* in a meta-model based measurement framework.
- Defining a series of experiments to see if the measurement framework can measure the amount of reuse for different software model types (Chapter 4 - the *assessment framework*).
- To ensure that the experiments use the measurement framework to test hypotheses, *meta-level components* in the measurement framework are linked to hypotheses in chapter 1 for each experiment (*methodology phase)* in the assessment framework. Meta-level components evaluated in a methodology phase are referred to as *assessed components*.
- To ensure that the experiments test the hypotheses:
  - A set of sub-hypotheses are defined for each hypothesis (section 4.4).
  - Hypotheses and sub-hypotheses tested in a methodology phase are identified as *hypotheses tested.*
  - Hypothesis measures are defined for each hypothesis tested in each methodology phase to determine the level of support for hypotheses. These are referred to as *hypothesis measures*.
  - Hypothesis measures are also defined for object-oriented and UML software models to detect the level of support for hypotheses using these software model types. This also verifies that object-oriented and UML software models are used in the experiments.
  - How results for these measures support any hypothesis tested are defined for each methodology phase. These are referred to as *assessment criteria.*
- The results of the experiments are the subject of Chapter 5. Results are summarised in *analysis reports* for each methodology phase and each hypothesis.
- Conclusions and any issues for further research are the subject of Chapter 6.

In this way coverage of the problem statement can be illustrated by showing the links between research questions, hypotheses, and sub-hypotheses (Table 4-1). Use of the measurement framework to test the hypotheses can be illustrated by naming the assessed components and hypotheses tested in each methodology phase (Table 4-2, Table 4-3, Table 4-4, and Table 4-5).

Table 4-1: Links between Research Questions, Hypotheses, and Null Hypotheses

Research Question	Hypothesis	Null Hypothesis	Sub-Hypotheses
RQ-1	H1	H0a	SH 1.1 to SH 1.12
RQ-2	H2	H0b	SH 2.1 to SH 2.12

It is important to distinguish between hypothesis measures and the measures for the amount of reuse. Hypothesis measures in the assessment framework (chapter 4) must not be confused with the measures for the amount of reuse described in the measurement framework (chapter 3). Hypothesis measures are indicators derived from the results of the experiments, that is, the methodology phases. The hypothesis measures are designed to measure the validity or support for the hypotheses.

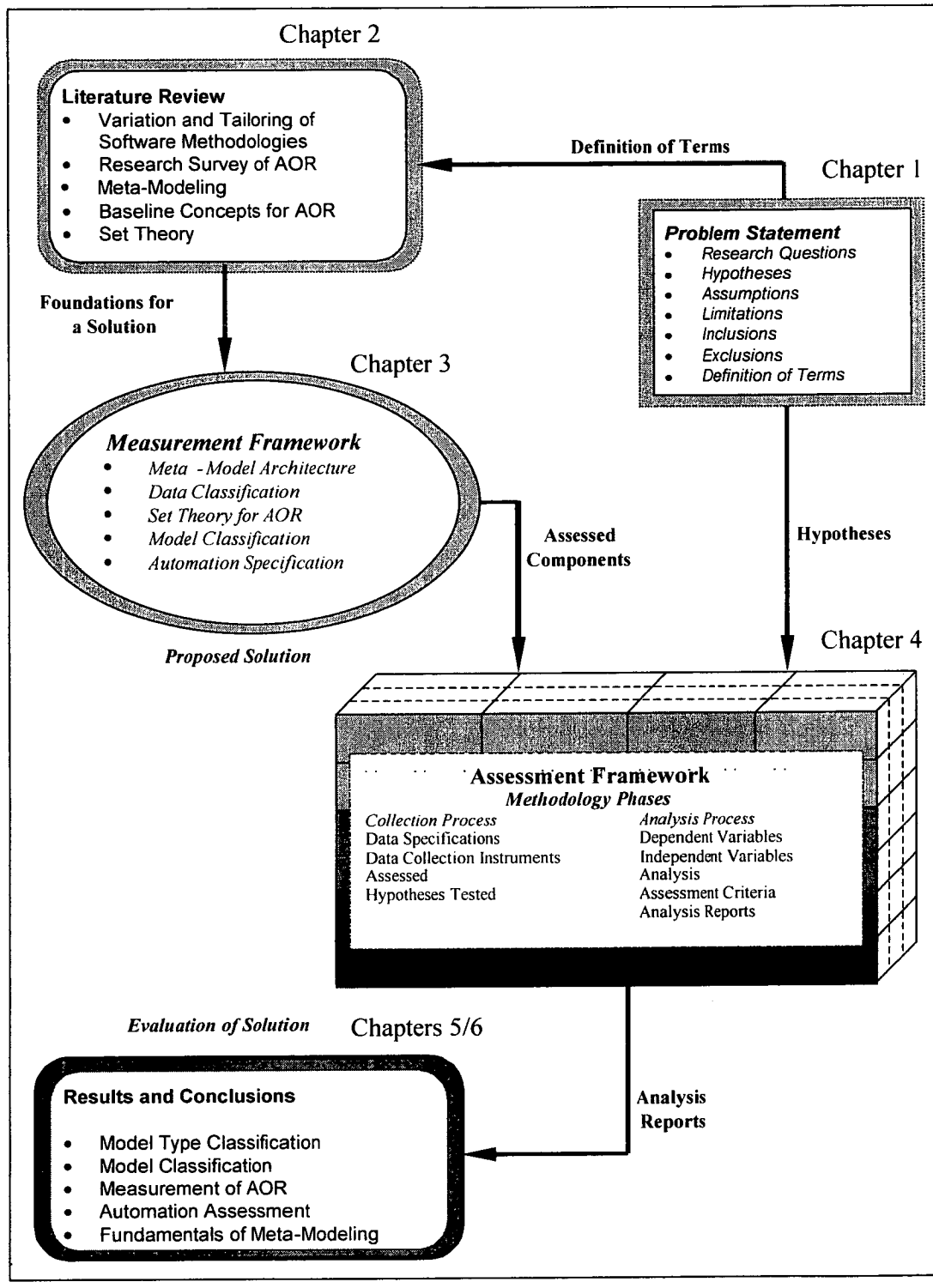

Figure 4-1: *Thesis Methodology and its relation to Thesis Chapters*

Table 4-2: Hypotheses Tested and Assessed Components for Software Model Type Classification Methodology Phase

Hypotheses Tested	Sub-Hypotheses Tested	Assessed Components
H0a	N/A	• Amount of Reuse Measurement M2 Model MLC. • Software Model Type Classification MLC. • Software Model Type Set Theory MLC. • Amount of Reuse Measurement M1 Model Specifier MLC
H0b	N/A	
H1	SH 1.1	
H2	SH 2.1	

Table 4-3: Hypotheses Tested and Assessed Components for Software Model Classification Methodology Phase

Hypotheses Tested	Sub-Hypotheses Tested	Assessed Components
H0a	N/A	• The Amount of Reuse Measurement M1 Model Specifier MLC. • The Software Model Classification MLC. • The Software Model Set Theory MLC. • The Amount of Reuse Measurement Model Specifier MLC.
H0b	N/A	
H1	SH 1.2 - SH 1.4	
H2	SH 2.2 - SH 2.4	

Table 4-4: Hypotheses Tested and Assessed Components for Measurement Testing Methodology Phase

Hypotheses Tested	Sub-Hypotheses Tested	Assessed Components
H0a	N/A	• The Amount of Reuse Measurement M1 Model Specifier MLC. • The Software Model Classification MLC. • The Software Model Set Theory MLC. • The Amount of Reuse Measurement Model Specifier MLC.
H0b	N/A	
H1	SH 1.5 - SH 1.12	
H2	SH 2.5 - SH 2.12	

Table 4-5: Hypotheses Tested and Assessed Components for Automation Assessment Methodology Phase

Hypotheses Tested	Sub-Hypotheses Tested	Assessed Components
H0b	N/A	• The TDL MLC at meta-level 5. • The TDL MLC at meta-level 4. • The M3 Model Specifier MLC. • The UML MLC at meta-level 4. • The Generic M3 Model MLC. • The UML MLC at meta-level 3. • The TDL MLC at meta-level 3. • The Set Theory MLC at meta-level 3. • The Measurement Model Specifier MLC. • The Amount of Reuse Measurement M2 Model MLC. • The Software Model Type Set Theory MLC. • The Software Model Type Classification MLC. • The Amount of Reuse Measurement M1 Model Specifier MLC. • The UML MLC at meta-level 2. • The TDL MLC at meta-level 2. • The Set Theory MLC at meta-level 2. • The Software Model Set Theory MLC. • The Software Model Classification MLC. • The Amount of Reuse Measurement Model Specifier MLC. • The Measurement Data Classification MLC.
H2	SH 2.1 - SH 2.12	

4.3 Criteria for Success – Claims Made

The claims made about the measurement framework in Chapter 3 are:

Claim 1. It can classify a range of different software model types.

Claim 2. It can use the software model type classifications to classify different software models and measure the size of these models in a predictable and consistent way.

Claim 3. It can use the software model type classifications to measure the amount of reuse of different software models in a predictable and consistent way.

Claim 4. It is a general framework, i.e. it can perform 1,2, and 3 using a common set of components. This implies that the automated version uses a common set of implementation components to do 1, 2, and 3.

The four claims naturally lead to four experiments to test these claims. These claims (1- 4) are closely related to the methodology phases of the assessment framework. So the experiments needed are:

1. One experiment to see if the measurement framework can classify different software model types. This experiment is referred to as the ***software model type classification*** methodology phase in the assessment framework (See Appendix C2 for the guide to interpreting hypothesis measures).
2. One experiment to see if the measurement framework can use the software model type classifications to classify different software models that are based on different software model types and measure the size of these software models. This experiment is referred to as the ***software model classification*** methodology phase (See Appendix C3 for the guide to interpreting hypothesis measures).
3. One experiment to see if the measurement framework can use the software model type classifications the measure the amount of reuse for different software models that are based on different software model types. This experiment is referred to as the ***measurement testing*** methodology phase (See Appendix C4 for the guide to interpreting hypothesis measures).
4. One experiment to see of the measurement framework uses a common set of components to do the previous three experiments. This experiment is referred to as the ***automation assessment*** methodology phase (See Appendix C5 for the guide to interpreting hypothesis measures).

In doing the four experiments two things are verified. Firstly, the experiments evaluate the measurement framework against the claims made.

Secondly, the hypotheses are tested because the measurement framework:

- Is based on meta-modelling.
- Is being evaluated to see if it can measure the amount of reuse for different kinds of software models (experiments 1,2, and 3), and
- Is being evaluated to determine if it is a general framework (experiment 4).

The next section revisits the hypotheses.

4.4 Hypotheses and Sub-Hypotheses

To make the reading of the rest of the chapter more cohesive the hypotheses are restated here. The hypotheses were broken down into sub-hypotheses to break down the problem of evaluation into smaller problems and to ensure that the experiments test the hypotheses more systematically.

There are four claims made in section 4.3 and two of these claims (claim 3 and claim 4) match well with the hypotheses and the problem statement. However, claim 1 and claim 2 do not because they don't address directly the need to measure the amount of reuse for different kinds of software models, but they are important because they support measurement of the amount of reuse. So how to can all the claims and associated experiments be aligned with the testing of the hypothesis? The answer lies in defining a set of sub-hypothesis for hypotheses H1 and H2 that are designed to look at a specific claim in section 4.3 and still be relevant to the main hypothesis.

In the first experiment (***software model type classification***), there is a need to see if the measurement framework can classify different software model types. This leads to one sub-hypothesis for each hypothesis (SH 1.1 for H1, and SH 2.1 for H2).

In the second experiment (***software model classification***), there is a need to consider classification of different software models and measurement of their size. There are actually two dimensions to measurement of size in terms of the outcomes for the experiment.

1. Measurement of size with respect to different software model types (Tests for different software models based on the same software model type).
2. Measurement of size with respect to different software models (Tests for different software models for various software model types)

This suggests three additional sub-hypothesis for each hypothesis. Two sub-hypotheses for classification of different software models based on their type (SH 1.2 for H1, SH 2.2 for H2), two sub-hypotheses for measurement of size for different

software model types (SH 1.3 for H1, SH 2.3 for H2), and two sub-hypotheses for measurement of size for software models (SH 1.4 for H1, SH 2.4 for H2).

Sub-hypotheses also afford another opportunity to break down the problem of testing a hypothesis based on the reuse approach and the software model types tested. There are three things to consider for the third experiment (***measurement testing***).

1. The reuse approach used (internal composition, external composition, internal generation, external generation).
2. Measurement of the amount of reuse with respect to different software model types (Tests for different software models based on the same software model type).
3. The outcome with respect to different software models (Tests for different software models for various software model types)

This yields 4 × 2 = 8 sub-hypothesis for each hypothesis. Four sub-hypothesis for each hypothesis consider the measurement of reuse for different software model types based on the reuse approach (SH 1.5 – SH 1.8 for H1, SH 2.5 – SH 2.8 for H2). Another four sub-hypothesis consider the measurement of reuse for different software models based on the reuse approach (SH 1.9 – SH 1.12 for H1, SH 2.9 – SH 2.12 for H2).

Since the main concern of the fourth experiment (***automation assessment***) is concerned identifying limitations for the first three experiments, the sub-hypothesis for H2 can be used here without any additional sub-hypothesis.

It is now possible to link each research question to one or more hypotheses (**H1 – H2**). Each hypothesis is also liked to one of the null hypotheses (**H0a, H0b**). Table 4-1 illustrates the relationship between the set of sub-hypotheses introduced in section 4.3 and hypotheses **H1** and **H2**.

There is a mutual exclusivity between the null hypotheses **H0a, H0b** and hypotheses **H1** and **H2**. Each hypothesis has an associated null hypothesis part. If the null hypothesis is demonstrated to be false because that part of it is false, then this supports the truth of the hypothesis in the same row in Table 4-1. For example, if **H0a** is false, then this supports the truth of **H1**.

The hypotheses are repeated below

H0a: A framework based on meta-modelling cannot support measurement of the amount of reuse for any kind of software model.

H0b: A framework based on meta-modelling does not have any limitations in measurement of the amount of reuse with different kinds of software models.

H1: A framework based on meta-modelling that supports measurement of the amount of reuse for different kinds of software models.

H2: A framework based on meta-modelling has limitations in measurement of the amount of reuse with different kinds of software models.

Below are the sub-hypotheses for H1 and H2.

Sub-Hypotheses for H1

SH 1.1: A framework based on meta-modelling can classify different kinds of software models.

SH 1.2: A framework based on meta-modelling can classify different software models based on their type.

SH 1.3: A framework based on meta-modelling can measure size for different kinds of software models.

SH 1.4: A framework based on meta-modelling can measure size for different software models.

SH 1.5: A framework based on meta-modelling can measure the amount of reuse for internal composition reuse for different kinds of software models.

SH 1.6: A framework based on meta-modelling can measure the amount of reuse based on external composition reuse for different kinds of software models.

SH 1.7: A framework based on meta-modelling can measure the amount of reuse for internal generation reuse for different kinds of software models.

SH 1.8: A framework based on meta-modelling can measure the amount of reuse for external generation reuse for different kinds of software models.

SH 1.9: A framework based on meta-modelling can measure the amount of reuse for internal composition reuse for different software models.

SH 1.10: A framework based on meta-modelling can measure the amount of reuse for external composition reuse for different software models.

SH 1.11: A framework based on meta-modelling can measure the amount of reuse for internal generation reuse for different software models.

SH 1.12: A framework based on meta-modelling can measure the amount of reuse for external generation reuse for different software models.

Sub-Hypotheses for H2

SH 2.1: A framework based on meta-modelling that can classify different kinds of software models has limitations.

SH 2.2: A framework based on meta-modelling that can classify different software models based on their type has limitations.

SH 2.3: A framework based on meta-modelling that can measure size for different kinds of software models has limitations.

SH 2.4: A framework based on meta-modelling that can measure size for different software models has limitations.

SH 2.5: A framework based on meta-modelling that can measure the amount of reuse for internal composition reuse for different kinds of software models has limitations.

SH 2.6: A framework based on meta-modelling that can measure the amount of reuse based on external composition reuse for different kinds of software models has limitations.

SH 2.7: A framework based on meta-modelling that can measure the amount of reuse for internal generation reuse for different kinds of software models has limitations.

SH 2.8: A framework based on meta-modelling that can measure the amount of reuse for external generation reuse for different kinds of software models has limitations.

SH 2.9: A framework based on meta-modelling that can measure the amount of reuse for internal composition reuse for different software models has limitations.

SH 2.10: A framework based on meta-modelling that can measure the amount of reuse for external composition reuse for different software models has limitations.

SH 2.11: A framework based on meta-modelling that can measure the amount of reuse for internal generation reuse for different software models has limitations.

SH 2.12: A framework based on meta-modelling that can measure the amount of reuse for external generation reuse for different software models has limitations.

4.5 Dimensions of the Assessment Framework

The assessment framework has three essential dimensions illustrated in Figure 4-2. These are the methodology phases, the reuse approach, and the software model types and implementations.

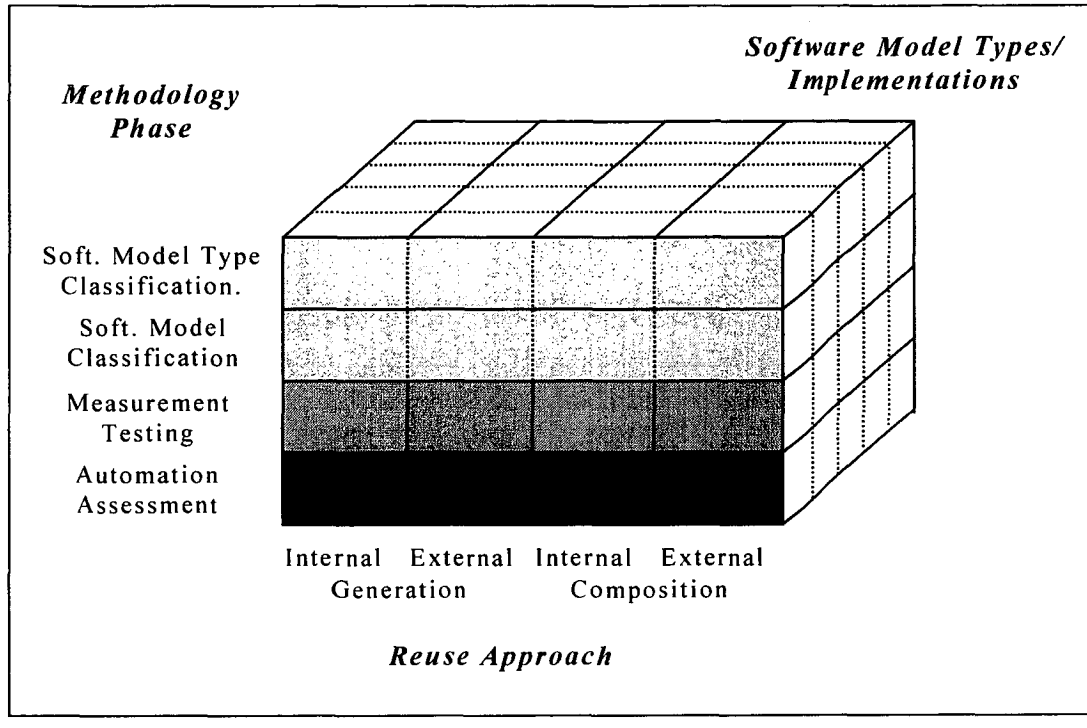

Figure 4-2: *Framework for Assessment of Theory (The Assessment Framework)*

Section 4.3 identified four experiments required to evaluate the measurement framework. These experiments are referred to as **methodology phases** and comprise the first dimension of the assessment framework.

From the baseline concepts for measurement of reuse in section 2.6 the *approach to reuse* and the *development scope of reuse* provide four major categories of reuse. The four categories are internal composition reuse, external composition reuse, internal generation reuse, and external generation reuse. The approach to reuse and development scope of reuse are combined to form the second dimension of the assessment framework. This dimension is referred to as the **reuse approach**.

Section 1.1 identifies the need to test the measurement framework using a range of software model types. This is the third dimension of the assessment framework referred to as **software model types & implementations**. Implementations refers to CASE tools that automate the modelling process for software model types.

Methodology Phases Dimension

These are the experiments in the study that are designed to test one or more hypotheses. The methodology phases are designed to reflect a process of how to answer the research questions using the hypotheses related to them (Table 4-1). The methodology phases are:

Software Model Type Classification is designed to see if the measurement framework classify different kinds of software models (claim 1).

Software Model Classification is designed to see if the measurement framework can classify different software models, and measure the size of different software models (claim 2).

Measurement Testing is designed to see if the measurement framework can measure the amount of reuse for different software models (claim 3).

Automation Assessment is designed to see if the measurement framework is a general framework (claim 4).

Reuse Approach Dimension

The reuse approach is considered along with the measures for the amount of reuse for the approach to reuse. The approach to reuse is either *internal composition*, *external composition*, *internal generation*, or *external generation*. These were described previously in section 3.5 and 2.6. This dimension is really only applicable to two methodology phases (measurement testing and automation assessment). But it is included as a dimension because the purpose of this thesis is evaluate a measurement framework to see if it can measure the amount of reuse. This makes the reuse approach an essential feature of any set of experiments required to test the hypotheses.

Associated with the approach are the measures for the amount of reuse. These are:

- *sms (source model size)*, *gms (generated model size)*, *cms (complete model size)*, *ag (amount generated)*, *ang (amount not generated)*, *wg (waste generated)*, *pccm (percentage contributuon to complete model)*, *png (percentage not generated)*, and *pwg (percentage waste generated)* for generation reuse.
- *lms (library model size)*, *ams (application model size)*, *ar (amount reused)*, *aa (amount added)*, *anr (amount not reused)*, *pram(percent reused in application model)*, *paam (percent added in application model)*, and *plmnr (percent library model not reused)* for composition reuse.

These were described previously in section 3.6. Internal and external variations are tested by setting the membership of the software models to the same project for internal reuse, or different projects for external reuse. Measures are then applied for either composition reuse or generation reuse as was discussed in section 3.5.

Software Model Types & Implementations Dimension

All software model types and implementations that automate them (CASE tools) that were tested in a methodology phase comprise this dimension. The first two dimensions remain the same regardless of the number of software model types used to evaluate the measurement framework. This dimension expands as more and more software model types are used to test the measurement framework according to the claims made in section 4.3. No software model types and implementations are prescribed as they are really part of the data gathering process. However, findings from chapter 2 (Section 2.5) necessitated selection of at least some software model types based on UML and the object-oriented paradigm.

Thus, each methodology phase considered the following elements:

- The hypotheses tested (**H0 – H2**) with the appropriate hypothesis measures.
- The reuse approach (*internal composition*, *external composition*, *internal generation*, *external generation*)
- The measures for the amount of reuse associated with the reuse approach (*sms*, *gms*, *cms*, *ag*, *ang*, *wg*, *pccm*, *png*, and *pwg* for generation reuse. *lms*, *ams*, *ar*, *aa*, *anr*, *pram*, *paam*, and *plmnr* for composition reuse, *size* for Software Model Classification).
- The Software Model Types with Implementations that were used in the methodology phase.

Each methodology phase must test some part of the measurement framework (one or more meta-level components) for its ability to satisfy one or more hypotheses. These are referred to as assessed components. Table 4-2, Table 4-3, Table 4-4, and Table 4-5 illustrate this for each methodology phase.

Now comes the need to define the experiments. This is the subject of section 4.6.

4.6 Methodology Phase Specification

Requirements for Experiments

The following questions need to be resolved when defining the experiments.

1. How can the experiments be described systematically to ensure they are executed properly and can be repeated?
2. How can the experiment be defined so that they test the hypotheses?
3. How can the experiments be defined so that they test the hypotheses using the measurement framework?
4. How can the experiments be defined so that they evaluate the measurement framework against the claims in section 4.3?

All of this is done using hypothesis measures and the methodology phase description template. As follows:

1. To define experiments more systematically and make them repeatable, the methodology phase description template sets out a number of sub-headings and required information for each sub-heading. This includes:
   - A description of how to conduct the experiment under a major subheading called *Collection Process*,
   - A description of the variables identified in the experiment under a major sub-heading called *Variables*, and
   - A description of how to analyse and document results under a major sub-heading called *Analysis Process*.
2. To ensure that the experiments test the hypotheses, each methodology phase must
   - Identify the hypotheses tested under a sub-heading *Hypotheses Tested*,
   - Define a series of hypotheses measures that quantify results of the experiments under a sub-heading called *Hypothesis Measures*, and
   - Define how results or values of the hypothesis measure indicate support for the hypotheses tested under a sub-heading called *Assessment Criteria*.
3. To ensure that the measurement framework is used to test the hypotheses, each methodology phase must identify the meta-level components assessed in the methodology phase under a sub-heading called *assessed components*.

4. To ensure that the experiments evaluate the measurement framework against the claims in section 4.2, each methodology phase must define and quantify the independent and dependent variables under the sub-headings of *Independent variables* and *dependent variables* to shed light on the claims made in section 4.2.

At this point it is worth emphasising the importance of the analysis process and variables, especially the hypothesis measures and assessment criteria. Rather than speculate after execution of an experiment of how results support hypotheses tested and claims made in section 4.3, better to speculate before execution of an experiment of how to measure support for the hypotheses and claims made so that we have a clear idea of how an experiment needs to be designed to ensure that it does test the hypotheses and evaluate the measurement framework against the claims made in section 4.3.

Methodology Phase Description Template

A complete description of the methodology phase description template can be found in Appendix C1. Descriptions of the methodology phases in this chapter use the same headings. The structure of this template is illustrated in Figure 4-3. Each phase has two basic processes and a set of variables. The processes are the collection process followed by the analysis process. The variables consist of independent and dependent variables.

The Collection Process: The collection process describes the documentation and procedures followed to collect and/or use data in the experiment.

The Variables: Both independent and dependent variables represent the link between the collection process and analysis process. Variables are identified from data gathered in the collection process and are used to assist analysis of a specific methodology phase according to the claims in section 4.3.

The Analysis Process: The analysis process describes how data gathered or used during the collection process is analysed to test hypotheses. This is based on hypothesis measures and assessment criteria.

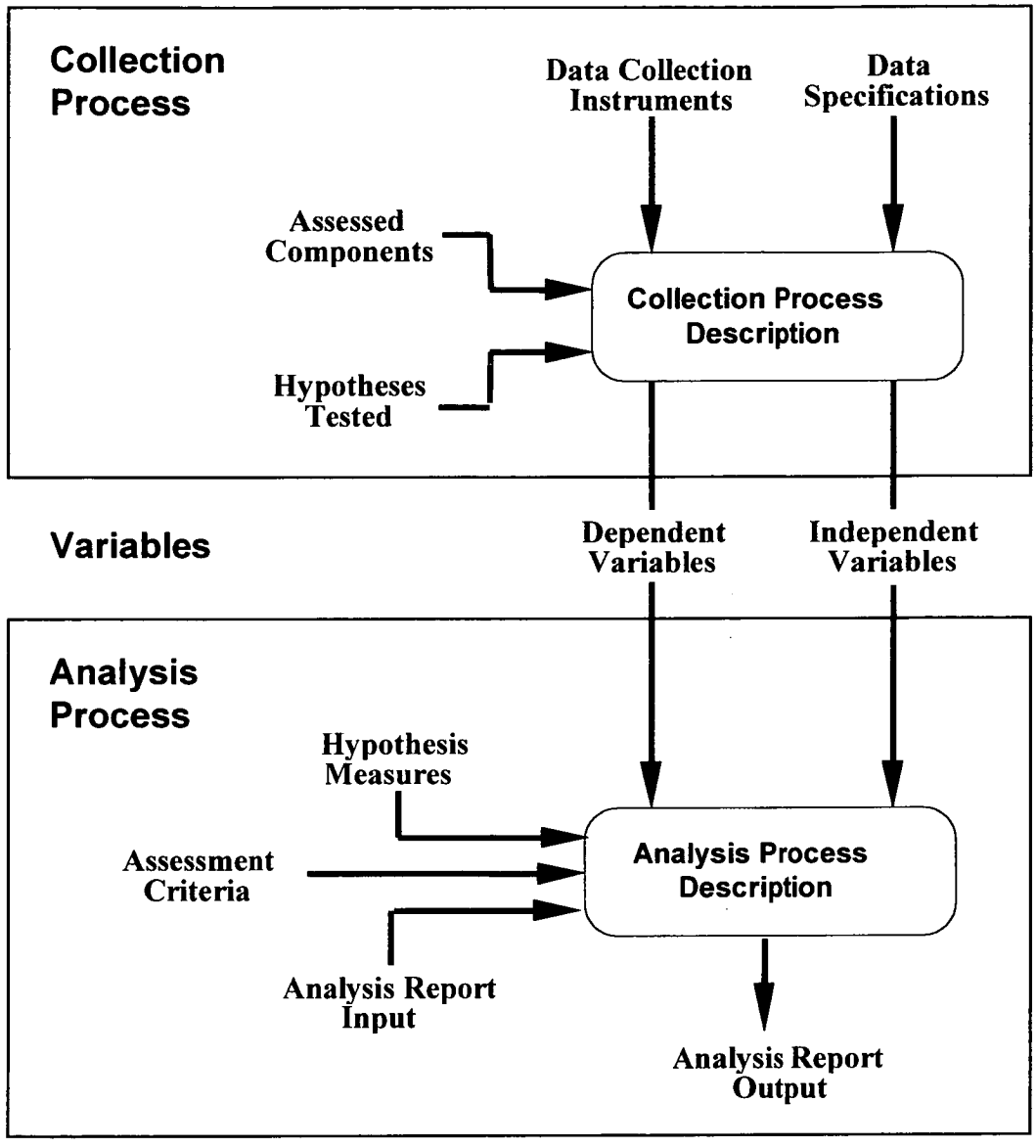

Figure 4-3: *Template for description of methodology phases*

Each phase starts with a heading that is the name of the methodology phase defined in the assessment framework. A *Methodology Phase Identifier* (MP_ID) is attached as a suffix to the name of the methodology phase in the heading. The MP_ID is used as an abbreviated reference to the methodology phase. A short description of the purpose for the methodology phase is given under this heading. Figure 4-4 illustrates an example of the heading for a methodology phase.

Software Model Type Classification: SMTC

This phase is designed to provide the necessary data for assessing the measurement framework's classification of different software model types.

Figure 4-4: *Specification of methodology phase heading using the template*

The collection process, variables, and analysis process are described under separate subheadings. Each methodology phase has the following structure for headings.

Phase Name: MP_ID

- **Collection Process**
  - **Assessed Components**
  - **Hypotheses Tested**
  - **Data Specifications**
  - **Data Collection Instruments**
  - **Collection Process Description**
- **Variables**
  - **Independent Variables**
  - **Dependent Variables**
- **Analysis Process**
  - **Hypothesis Measures**
  - **Assessment Criteria**
  - **Analysis Report Input**
  - **Analysis Report Output**
  - **Analysis Process Description**

Collection Process

Under the subheading Collection Process are five other subheadings that define the collection process. The sub-headings are Assessed Components, Hypotheses Tested, Data Specifications, Data Collection Instruments, and Collection Process Description.

Assessed Components: One of more MLC's from the meta-model architecture for measurement of the AOR are named under this subheading. The components are assessed according to the assessment criteria in the methodology phase.

Hypotheses Tested: Names the hypotheses tested. Sub-hypotheses tested and any null hypotheses parts are also included.

Data Specifications: Any relevant data used prior to execution of the collection process description.

Data Collection Instruments: The format of documentation for collection of data during execution of the collection process description. Data collected using a data collection instrument for one methodology phase can become data specifications for another methodology phase.

Collection Process Description: A step by step description of the procedure of how the data is collected and documented using the data specifications and data collection instruments.

Variables

Under the subheading Variables are two subheadings that define the variables. These are Independent Variables and Dependent Variables.

Independent Variables: Data specifications, data collected using data collection instruments, and assessed components that are independent variables for this phase. Any measures for these variables are also described.

Dependent Variables: Data collected using data collection instruments that are dependent variables for this phase. Any measures for these variables are also described.

Analysis Process

Under the subheading Analysis Process are five subheadings that define the analysis process. These are Hypothesis Measures, Assessment Criteria, Analysis Report Input, Analysis Report Output, and Analysis Process Description.

Hypothesis Measures: The hypothesis measures used to evaluate hypotheses identified under the previous heading "Hypotheses Tested". These consist of a number of measures, referred to as hypothesis measures, and their calculations. Hypothesis measures are divided into three categories as follows:

> ***Base indicators.*** A base indicator is a calculated value derived from the data specifications and/or data collection instruments in the methodology phase. By themselves these indicators are not used to measure support for hypotheses. They are compared with other indicators for analysis of data.
>
> ***Coverage indicators.*** A coverage indicator is a calculated value derived from data collection instruments and possibly data specifications. These are designed to quantify how much a hypothesis is supported based on the success or failure of the experiment conducted in the methodology phase. Their primary purpose is to test hypothesis H1.
>
> ***Limitation indicators.*** A limitation indicator is a calculated value derived from data collection instruments and possibly data specifications. These are designed to quantify the limitations of the measurement framework. Their primary purpose is to test hypotheses H2.

Note: Hypothesis measures are designed to help test the validity of hypotheses. In contrast, measures for variables are designed to help describe what happened after execution of the collection process. They give some indication of the answer to the research question associated with the methodology phase, but a not the rigorous answer required for testing of hypotheses.

Assessment Criteria: Describes how hypothesis measures are used to test the hypotheses. Results from a hypothesis measure indicate some level of support for it. Each hypothesis measure can either *support*, *not support*, or *deny* one or more hypotheses depending on the value obtained. A range of values for a hypothesis measure may also support, not support, or deny a hypothesis. In such cases, there may be an optimum value declared for the hypothesis measure.

For each hypothesis and its related sub-hypotheses, the relevant hypothesis measure from hypothesis measures is named and the value or range of values are defined that either support, do not support, or deny the truth of a hypothesis. Where applicable, optimum values are also identified.[1]

Analysis Report Input: The format of the documentation that is used during execution of the analysis process description. There are two kinds of analysis reports:

> ***Methodology Phase Analysis Report:*** This report is used to document the results after execution of a methodology phase. There is one report per methodology phase.
>
> ***Hypothesis Analysis Report:*** This report is used to document the results for a hypothesis and its sub-hypotheses or hypothesis parts after execution of each relevant methodology phase. There is one report per hypothesis.

Use of analysis reports in a methodology phase are described under the heading "Analysis Process Description".

Note: If the same hypothesis analysis report appears in more than one methodology phase description it is the same report. This report is being modified and updated by each methodology phase that affects it.

See the sections under the headings "Format of Hypothesis Analysis Reports" and "Format of Methodology Phase Analysis Reports" in Appendix C6 for more details on analysis reports.

Analysis Report Output: Describes the analysis reports either generated or modified after execution of the analysis process description. The structure of the analysis

---

reports is from the analysis report input. The modification or generation of the analysis reports is based on the analysis process description.

Analysis Process Description: A step by step description of the procedure of how independent variables and dependent variables are analysed using the hypothesis measures, analysis reports used, and assessment criteria to produce the analysis report output.

The next four sections are descriptions of the methodology phases based on the methodology phase description template.

4.7 The Methodology Phase Descriptions

Software Model Type Classification: SMTC

Experiment Overview

In section 4.3 the first claim made was that the measurement framework can classify different software model types. This methodology phase is an experiment designed to test if this is true. The experiment can be summarised as follows:

Step 1. A range of software model types were identified from the literature.

Step 2. A selection of software model types from Step 1 that are implemented on CASE tools are used for testing the measurement framework.

Step 3. The measurement framework is used to see if it can classify the selection of software model types in Step 2 by comparing the expected software model type classification with the one classified using the measurement framework.

This is the essence of the collection process. There are three basic outcomes for any given software model type used in the experiment:

Outcome 1. The software model type is implemented on a CASE tool and the measurement framework classified the software model type successfully, i.e. that the measurement framework can arrive at a classification for the software model type. This supports hypothesis H1.

Outcome 2. The software model type is implemented on a CASE tool and the measurement framework cannot classify the software model type. This does not support hypothesis H1.

Outcome 3. The software model type is not implemented on a CASE tool and the experiment cannot determine if the measurement framework can classify the software model type. This supports hypothesis H2.

Based on the above outcomes, support for H1 and H2 can be quantified as follows:

- H1 is supported by the number of software model types successfully classified by the measurement framework.
- H1 is not supported by the number of software model types not classified by the measurement framework.
- H2 is supported by the number of software model types identified but not implemented on a CASE tool and therefore not used in the experiment.

The three basic quantities above are the essence of he hypothesis measures and assessment criteria in this methodology phase. Now follows a description of the methodology phase in more detail.

Collection Process

Assessed Components

The assessed components in the framework for this phase are:

1. The Amount of Reuse Measurement M2 Model MLC.
2. The Software Model Type Classification MLC.
3. The Software Model Type Set Theory MLC.
4. The Amount of Reuse Measurement M1 Model Specifier MLC.

Hypotheses Tested

The hypotheses tested are:

H0a and **H0b**.

H1 via **SH 1.1**.

H2 via **SH 2.1**.

H0a: A framework based on meta-modelling cannot support measurement of the amount of reuse for any kind of software model.

H0b: A framework based on meta-modelling does not have any limitations in measurement of the amount of reuse with different kinds of software models.

H1: A framework based on meta-modelling supports measurement of the amount of reuse for different kinds of software models.

SH 1.1: A framework based on meta-modelling can classify different kinds of software models.

H2: A framework based on meta-modelling has limitations in measurement of the amount of reuse with different kinds of software models.

SH 2.1: A framework based on meta-modelling that can classify different kinds of software models has limitations.

Data Specifications

There are no data specifications for this phase although literature sources from the literature review were used to help identify a number of software model types and their respective implementations.

Data Collection Instruments

Data collected was documented using the **Software Model Types** table, the **Implementations** table, the **Software model types and Implementations** table, and the **Software model type classifications** table.

The **Software Model Types** table contains the name of each software model type and also the literature sources used to identify it.

The **Implementations** table names each implementation and the vendor that developed it.

The **Software model types and Implementations** table links software model types to implementations. Each row is given a unique id, called the **SMTI ID**.

The **Software model type classifications** table contains one row for each software model type used to assess the framework components. Each row contains the **expected classification** and **actual classification** scheme for a software model type.

Collection Process Description

1. A selection of software model types are named and linked to one or more literature sources using the **Software model types** table.
2. A selection of implementations are named and linked to a vendor using the **Implementations** table.
3. Each software model type is linked to an implementation using the **Software model types and implementations** table. One row for each link.
4. For each row in the **Software model type and implementations** table, a new row is made in the **Software model type classifications** table.
5. In the **Software model type classifications** table the **expected classification** is defined without using the automated version of measurement framework.
6. The results of classifying the software model type is documented in one row of the **Software model type classifications** table by entering the **actual classification** defined using the automated version of the measurement framework.

Variables

Independent Variables

The following independent variables were obtained after execution of the collection process for the software model type classification phase.

Independent variables consist of:

1. The **Software model types** table.
2. The **Implementations** table.
3. The **Software model types and implementations** table.
4. The **Expected classification scheme, SMTI ID,** and **Model Type ID** for each row in the **Software model type classifications** table.
5. The Amount of Reuse Measurement M2 Model MLC.
6. The Software Model Type Classification MLC.
7. The Software Model Type Set Theory MLC.
8. The Amount of Reuse Measurement M1 Model Specifier MLC.

Dependent Variables

The following dependent variables were obtained after execution of the collection process for the software model type classification methodology phase.

Dependent variables consist of:

1. The **Actual classification scheme** for each row in the **Software model type classifications** table.
2. The MLC instances of the Amount of Reuse Measurement M2 Model MLC.
3. The MLC instances of the Software Model Type Classification MLC.
4. The MLC instances of the Software Model Type Set Theory MLC.
5. The MLC instances of the Amount of Reuse Measurement M1 Model Specifier MLC.

Analysis Process

Hypothesis Measures

Base indicators, coverage indicators, and limitation indicators used in this phase are tabulated in Table 4-6, Table 4-7, and Table 4-8. See Appendix C2 for a guide to interpretation of all hypothesis measures used in this methodology phase.

Table 4-6 lists the base indicators used in the software model type classification methodology phase. These include the number of software model types available and the number of software model types linked to implementations. These were used to support the calculations of coverage and limitation indicators in the software model type classification methodology phase.

Table 4-6: Base Indicators used in Software Model Type Classification

Model Type Category	Names of Measures
All Software Model Types	**SMT_BI, SMTI_BI, SMTC_BI**
Object-oriented/UML Software Model Types	**OU_SMT_BI, OU_SMTI_BI, OU_SMTC_BI**

Below are two examples of base indicators from the guide to interpretation (Appendix C2).

Name of Measure	Calculation
SMT_BI (Software Model Types Base Indicator)	#rows in the **Software model types** table
SMTC_BI (Software Model Type Classifications Base Indicator)	#rows in the **Software model type classifications** table

SMT_BI: This indicator is the number of software model types identified for the purposes of this thesis and calculated as a number of rows in the **Software model types** table.

SMTC_BI: This indicator is the number of software model type classifications in the **Software model type classifications** table. The indicator represents the number of attempts made classification of software model types with implementations.

Table 4-7 lists the coverage indicators used in the software model type classification methodology phase.

Table 4-7: Coverage Indicators used in Software Model Type Classification

Model Type Category	Names of Measures
All Software Model Types	**Category 1: CC_CI, SC_CI, PSC_CI** **Category 2: UC_CI, PUC_CI**
Object-Oriented/UML Software Model Types	**Category 1: OU_CC_CI, OU_SC_CI, OU_PSC_CI** **Category 2: OU_UC_CI, OU_PUC_CI**

Category 1 indicators measure the degree or level of successful classification of software model types.

Category 2 indicators measure the level of failure or unsuccessful classification of software model types.

Below are two examples of coverage indicators from the guide to interpretation (Appendix C2). The first example is a category 1 indicator that quantifies outcome 1 stated in the experiment summary. The second example is a category 2 indicator that quantifies the outcome 2 stated in the experiment summary.

Name of Measure	Calculation
SC_CI (Successful Classification Coverage Indicator)	#rows where **Expected classification scheme** = **Actual classification scheme** in the **Software model type classifications** table
UC_CI (Unsuccessful Classification Coverage Indicator)	#rows where **Actual classification scheme** = "NOT ABLE TO CLASSIFY SOFTWARE MODEL TYPE" in the **Software model type classifications** table

SC_CI: This indicator attempts to express the degree of successful classification relative to the number of classification attempts. The value is obtained by counting the number of successful classification attempts. This value is compared with SMTC_BI. A low value indicates that the degree of successful classification is low. The lowest possible value for SC_CI is zero. A high value indicates that the degree of successful classification is high. The highest possible value is SMTC_BI.

UC_CI: This indicator attempts to express the degree of unsuccessful classification relative to the number of classification attempts. The value is obtained by counting the number of unsuccessful classification attempts. This value is compared with SMTC_BI. A low value indicates that the degree of unsuccessful classification is low. The lowest possible value for UC_CI is zero. A high value indicates that the

degree of unsuccessful classification is high. The highest possible value is SMTC_BI.

Table 4-8 lists the limitation indicators used in the software model type classification methodology phase.

Table 4-8: Limitation Indicators used in Software Model Type Classification

Model Type Category	Names of Measures
All Software Model Types	**Category 1: PCSMTI_LI, CSMTI_LI** **Category 2: SCSMT_LI, PSCSMT_LI, SCSMTI_LI, PSCSMTI_LI** **Category 3: UCSMTI_LI, PUCSMT_LI, UCSMT_LI, PUCSMTI_LI**
Object-oriented/UML Software Model Types	**Category 1: OU_PCSMTI_LI, OU_CSMTI_LI** **Category 2: OU_SCSMT_LI, OU_PSCSMT_LI, OU_SCSMTI_LI OU_PSCSMTI_LI** **Category 3: OU_UCSMT_LI, OU_PUCSMT_LI, OU_UCSMTI_LI, OU_PUCSMTI_LI**

Category 1 indicators measure the possible coverage of software models types relative to the number of software model types classified.

Category 2 indicators measure the degree of successful classification of software model types relative to the number of software model types available for classification.

Category 3 indicators measure the level of failure or unsuccessful classification of software model types relative to the number of software model types available for classification.

Below are three examples of limitation indicators from the guide to interpretation (Appendix C2), one for each category. Each one of these indicators quantifies outcome 3 in the experiment overview.

Name of Measure	Calculation
PCSMTI_LI (Possible Coverage of Software Model Types with Implementations Limitation Indicator)	#rows in the **Software model types and Implementations** table where **Software model type name** = **Software model type name** in one row in the **Software model types** table
PSCSMT_LI (Percentage Successful Coverage of Software Model Types Limitation Indicator)	(#rows where **Expected classification scheme** = **Actual classification scheme** in the **Software model type classifications** table) / (#rows in **Software model types** table) * 100
PUCSMT_LI (Percentage Unsuccessful Coverage of Software Model Types Limitation Indicator)	(#rows where **Actual classification scheme** = "NOT ABLE TO CLASSIFY SOFTWARE MODEL TYPE" in the **Software model type classifications** table) / (#rows in **Software model types** table) * 100

PCSMTI_LI: This indicator attempts to assesses possible coverage of software model types relative to the number of software model types with implementations. The value is calculated by counting the number of software model types that also exist as software model types with implementations. The value is compared with SMT_BI. A high value indicates high possible coverage of software model types. The highest possible value for PSCSMTI_LI is equal to SMT_BI. A low value indicates low possible coverage of software model types. The lowest possible value for PSCSMTI_LI is 0.

PSCSMT_LI: This indicator attempts to express the degree of successful classification relative to software model types as a percentage. The value is calculated by dividing the number of successful classification by the number of software model types and multiplying the result by 100. A high value indicates a high degree of successful classification relative to software model types. A low value indicates a low degree of successful classification relative to software model types.

PUCSMT_LI: This indicator attempts to express the degree of unsuccessful classification relative to software model types as a percentage. The value is calculated by dividing the number of unsuccessful classification by the number of software model types and multiplying the result by 100. A high value indicates a high degree of unsuccessful classification relative to software model types. A low value indicates a low degree of unsuccessful classification relative to software model types.

Assessment Criteria

Hypothesis measures used to test the hypotheses is illustrated in:

- Table 4-9 for **H0a** and **H0b**.
- Table 4-10 for **H1/SH 1.1**.
- Table 4-11 for **H2/SH 2.1**.

Refer to Appendix C2 for further details on assessment criteria.

Table 4-9 names the hypothesis measures used for testing the null hypothesis **H0a** and **H0b**.

Table 4-9: Assessment of **H0a** and **H0b** using Hypothesis Measures

Hypothesis Part	Measures Used
H0a	PSCSMT_LI, PSCSMTI_LI, PSC_CI PUCSMT_LI, PUCSMTI_LI, PUC_CI
	OU_PSCSMT_LI, OU_PSCSMTI_LI, OU_PSC_CI OU_PUCSMT_LI, OU_PUCSMTI_LI, OU_PUC_CI
H0b	PCSMTI_LI, CSMTI_LI SCSMT_LI, PSCSMT_LI, SCSMTI_LI, PSCSMTI_LI UCSMT_LI, PUCSMT_LI, UCSMTI_LI PUCSMTI_LI
	OU_PCSMTI_LI, OU_CSMTI_LI OU_SCSMT_LI, OU_PSCSMT_LI, OU_SCSMTI_LI, OU_PSCSMTI_LI OU_UCSMT_LI, OU_PUCSMT_LI, OU_UCSMTI_LI, OU_PUCSMTI_LI

How the values for the hypothesis measures support H0 is detailed in the guide to interpretation (Appendix C2). Below are some examples from the guide to interpretation that illustrate this.

Hypothesis Part	Name of Measure	Value to Support Hypothesis	Value to Deny Hypothesis
H0a	**PSCSMT_LI**	PSCSMT_LI = 0	PSCSMT_LI > 0
	PUCSMT_LI	PUCSMT_LI = 100	PUCSMT_LI < 100
H0b	**PCSMTI_LI, SMT_BI**	PCSMTI_LI = SMT_BI and SMT_BI >=1	PCSMTI_LI < SMT_BI and SMT_BI >= 1 **OV:** PCSMTI_LI = 0

The truth of **H0a** is *supported* if:

- No successful classifications could be found for any software model type (PSCSMT_LI = 0).
- All attempts at classification found for any software model type failed (PUCSMT_LI = 100)

To *deny* **H0a**:

- At least one successful classification must be found for a software model type (PSCSMT_LI > 0),
- Not all classification attempts found for software model types were unsuccessful (PUCSMT_LI < 100)

The truth of **H0b** is *supported* if:

- The possible coverage of software model types is complete (PCSMTI_LI = SMT_BI) for each software model type (SMT_BI >= 1).

To *deny* **H0b**:

- The possible coverage of software model types must not be complete (PCSMTI < SMT_BI) for at least one software model type (SMT_BI >= 1).

Table 4-10 names the hypothesis measures used for testing the hypotheses **H1** via **SH 1.1**.

Table 4-10: Assessment of Hypotheses **H1** using hypothesis measures

Hypothesis	Sub-Hypotheses	Measures Used
H1	**SH 1.1**	**CC_CI, SC_CI, SCSMT_LI, SCSMTI_LI, PSCSMT_LI, PSCSMTI_LI, UCSMT_LI, PSC_CI UCSMTI_LI, PUCSMT_LI, PUCSMTI_LI, UC_CI, PUC_CI** **OU_CC_CI, OU_SC_CI, OU_PSC_CI, OU_SCSMT_LI, OU_SCSMTI_LI, OU_PSCSMT_LI, OU_PSCSMTI_LI OU_UCSMT_LI, OU_UCSMTI_LI, OU_PUCSMT_LI, OU_PUCSMTI_LI, OU_UC_CI, OU_PUC_CI**

How the values for the hypothesis measures support H1 is detailed in the guide to interpretation (Appendix C2). Below are some examples from the guide to interpretation that illustrate this.

Name of Measure	Value to Support Hypothesis	Value to Not Support Hypothesis
SC_CI, SMTC_BI	SC_CI >=2 and SMTC_BI > = 2 **OV:** SC_CI = SMTC_BI	SC_CI < 2 and SMTC_BI >= 2 **OV:** SC_CI = 0
UC_CI, SMTC_BI	UC_CI + 2 <= SMTC_BI and SMTC_BI >= 2 **OV:** UC_CI = 0	UC_CI + 2 > SMTC_BI and SMTC_BI >= 2 **OV:** UC_CI = SMTC_BI

(**OV:** Hypothesis Measure = Optimum Value refers to the optimum value for the hypothesis measure)

The truth of **H1** is *supported* if the truth of **SH 1.1** is *supported.*

To *support* **SH 1.1:**

- At least two classification attempts must be successful (SC_CI >= 2) with at least two attempts at classification (SMTC_BI >= 2)
- At least two classification attempts must not be unsuccessful (UC_CI + 2 <= SMTC_BI) with at least two attempts at classification (SMTC_BI >= 2)

SH 1.1 is *not supported* if:

- There are fewer than two successful classification attempts (SC_CI < 2) for at least two attempts at classification (SMTC_BI >= 2).
- There are fewer than two classification attempts that were not unsuccessful (UC_CI + 2 > SMTC_BI) for at least two attempts at classification (SMTC_BI >= 2).

Table 4-11 names the hypothesis measures used for testing the hypotheses **H2** via **SH 2.1**.

Table 4-11: Assessment of Hypotheses **H2** using hypothesis measures

Hypothesis	Sub-Hypotheses	Measures Used
H2	**SH 2.1**	**PCSMTI_LI, CSMTI_LI** **SCSMT_LI, PSCSMT_LI, SCSMTI_LI, PSCSMTI_LI** **UCSMT_LI, PUCSMT_LI, UCSMTI_LI, PUCSMTI_LI** **OU_PCSMTI_LI, OU_CSMTI_LI** **OU_SCSMT_LI, OU_PSCSMT_LI, OU_SCSMTI_LI, OU_PSCSMTI_LI** **OU_UCSMT_LI, OU_PUCSMT_LI, OU_UCSMTI_LI, OU_PUCSMTI_LI**

How the values for the hypothesis measures support H2 is detailed in the guide to interpretation (Appendix C2). Below are some examples from the guide to interpretation that illustrate this.

Name of Measure	Value to Support Hypothesis	Value to Not Support Hypothesis
PCSMTI_LI, SMT_BI	PCSMTI_LI < SMT_BI and SMT_BI >= 1 **OV:** PCSMTI_LI = 0	PCSMTI_LI = SMT_BI and SMT_BI >=1
PSCSMT_LI	PSCSMT_LI < 100 **OV:** PSCSMT_LI = 0	PSCSMT_LI = 100
PUCSMT_LI	PUCSMT_LI > 0 **OV:** PUCSMT_LI = 100	PUCSMT_LI = 0

(**OV:** Hypothesis Measure = Optimum Value refers to the optimum value for the hypothesis measure)

The truth of **H2** is *supported* if the truth of **SH 2.1** is *supported.*

To *support* **SH 2.1**:

- The possible coverage of software model types must not be complete (PCSMTI < SMT_BI) for at least one software model type (SMT_BI >= 1).
- Not all classification attempts relative to software model types were successful (PCSMT_LI < 100) and some of these failed (PUCSMT_LI > 0).

SH 2.1 is *not supported* if:

- The possible coverage of software model types is complete (PCSMTI_LI = SMT_BI) for each software model type (SMT_BI >= 1).
- All classification attempts were successful relative to software model types (PSCSMT_LI = 100) and no classification attempts failed (PUCSMT_LI = 0).

To summarise, **H0a/b** is *supported* if:

- all attempts at classification of software model types fail (**H0a**) and
- all software model types available were classified and all of these classification attempts are successful (**H0b**).

H0a/b, is *denied* if:

- at least one attempt at classification of a software model types is successful (**H0a**), or
- at least one software model type available is not classified or at least one attempt at classification of a software model type fails (**H0b**).

H1 is *supported* if at least two software model types are successfully classified.

H1 is *not supported* if at most one software model type is successfully classified.

H2 is *supported* if at least one software model type available is not classified or at least one attempt at classification of a software model type fails.

H2 is *not supported* if all software model types available are classified and all attempts at classification of software model types are successful.

This concludes the assessment criteria in the software model type classification methodology phase for hypotheses **H0a**, **H0b**, **H1/SH 1.1**, and **H2/SH 2.1**.

Analysis Report Input

The methodology phase analysis reports used in this phase are:

The Software Model Type Classification Analysis Report. See the section under the heading "Software Model Type Classification Report" in Appendix D1 for a complete example.

The hypothesis analysis reports used in this phase are:

The H0 Analysis Report. See the section under the heading "H0 Hypothesis Analysis Report" in Appendix D5 for a complete example.

The H1 Analysis Report. See the section under the heading "H1 Hypothesis Analysis Report" in Appendices D6 - D7 for a complete example.

The H2 Analysis Report. See the section under the heading "H2 Hypothesis Analysis Report" in Appendices D8 - D9 for a complete example.

Analysis Report Output

The analysis reports generated by this methodology phase are:

- **The Software Model Type Classification Analysis Report.** This report is generated by adding all measures for independent and dependent variables in the software model type classification methodology phase along with the values obtained. In addition, values for all hypothesis measures are added to the report.

The analysis reports modified in this methodology phase are:

The H0 Analysis Report.

The H1 Analysis Report.

The H2 Analysis Report.

The above hypothesis analysis reports include the values obtained for all hypothesis measures from the software model type classification methodology phase. Identification of the level of support for each hypothesis are also included and relevant totals for sub-hypotheses or hypothesis parts and the hypothesis are updated.

Analysis Process Description

1. Calculate the values of the measures derived from the independent variables and dependent variables for the software model type classification methodology phase.
2. Enter the names and values for these measures in Software Model Type Classification Methodology Phase Analysis Report.

3. Calculate the values of the base indicators, coverage indicators, and limitation indicators for hypothesis measures listed under hypothesis measures for the software model type classification methodology phase.
4. Enter the values of the hypothesis measures in the appropriate rows of the Software Model Type Classification Methodology Phase Analysis report.
5. Copy the values for the hypothesis measures in the Software Model Type Classification Methodology Phase Analysis Report into the appropriate hypothesis analysis reports. For each of theses rows enter the MP_ID in the actual value column as "SMTC" and add the required values to support and deny the hypothesis in their respective columns.
6. Update the sub-totals and totals in the **H0** hypothesis analysis report, the **H1** hypothesis analysis report, the **H2** hypothesis analysis report, and the Software Model Type Classification Analysis Report.

Software Model Classification: SMC

Experiment Overview

In section 4.3 the second claim made was that the measurement framework can use software model type classifications to classify different software models and measure the size of these models in a predictable and consistent way. This methodology phase is an experiment designed to test if this is true. The experiment can be summarised as follows:

Step 1. All software model types from the software model type classification methodology phase are included in this methodology phase. This includes software model types implemented on CASE tools and software model types not implemented.

Step 2. For each software model type classified in the software model type classification methodology phase, a range of software model classifications are made to test various facets of the software model type classification.

Step 3. For each of the software models in Step 2, an attempt is made to classify the software model using the measurement framework. To determine success or failure the actual classification made using the automated version of the measurement framework is compared to the expected classification specified for the software model.

Step 4. For each of the software models in Step 2, an attempt is made to measure the size of the software model using the measurement framework. To determine success or failure the actual size measured using the automated version of the measurement framework is compared to the expected size specified for the software model.

This is the essence of the collection process.

There are five basic outcomes for any given software model type used in the experiment. These outcomes are not all mutually exclusive.

Outcome 1. The software model type was successfully classified and it was possible to classify different software models and measure their size based on the software model type using the measurement framework and the software model type classification. This supports hypothesis H1.

Outcome 2. The software model type was successfully classified and it was not possible to classify different software models or measure their size based on the software model type using the measurement framework and the software model type classification. This does not support hypothesis H1.

Outcome 3. The software model type was never classified and it was not possible to determine if the measurement framework could classify different software models or measure their size based on the software model type. This supports hypothesis H2..

Outcome 4. Successful classification of software models and measurement of their size is dependent upon successful classification of their software model type. This supports hypothesis H2.

Outcome 5. The same software model type classification was used to classify different software model types used in the software model classification methodology phase. This does not support hypothesis H2.

Outcome 6. Different software model type classifications were used to classify different software model types used in the software model classification methodology phase. This supports hypothesis H2.

Based on the above outcomes, support for hypotheses H1 and H2 can be quantified as follows:

- H1 is supported by the number of software model type classifications used to successfully classify different software models and measure the size.
- H1 is not supported by the number of software model type classifications used to unsuccessfully classify different software models and measure the size.
- H2 is supported by the number of software model types not classified and therefore not used in the software model classification methodology phase.
- H2 is supported by the number of times a successful classification attempt of a software model and measurement of its size is associated with a successful classification of its software model type.

- H2 is supported by the number of software model types with different software model type classifications used to classify different software models and measure their size.
- H2 is not supported by the number of software model types with common software model type classifications used to classify different software models and measure their size.

The six basic quantities above are the essence of hypothesis measures and assessment criteria in this methodology phase. Now follows a description of the methodology phase in more detail.

Collection Process

Assessed Components

The assessed components in the framework for this phase are:

1. The Amount of Reuse Measurement M1 Model Specifier MLC.
2. The Software Model Classification MLC.
3. The Software Model Set Theory MLC.
4. The Amount of Reuse Measurement Model Specifier MLC.

Hypotheses Tested

The hypotheses tested were:

H0a.

H0b.

H1 via **SH 1.2**, **SH 1.3**, and **SH 1.4**.

H2 via **SH 2.2**, **SH 2.3**, and **SH 2.4**.

H0a: A framework based on meta-modelling cannot support measurement of the amount of reuse for any kind of software model.

H0b: A framework based on meta-modelling does not have any limitations in measurement of the amount of reuse with different kinds of software models.

H0: A framework based on meta-modelling that supports measurement of the amount of reuse for different kinds of software models does have advantages.

SH 1.2: A framework based on meta-modelling that can classify different software models based on their type does have advantages.

SH 1.3: A framework based on meta-modelling that can measure size for different kinds of software models does have advantages.

SH 1.4: A framework based on meta-modelling that can measure size for different software models does have advantages.

H1: A framework based on meta-modelling has limitations in measurement of the amount of reuse with different kinds of software models.

SH 2.2: A framework based on meta-modelling that can classify different software models based on their type has limitations.

SH 2.3: A framework based on meta-modelling that can measure size for different kinds of software models has limitations.

SH 2.4: A framework based on meta-modelling that can measure size for different software models has limitations.

Data Specifications

Data specifications used in this methodology phase are the **Software Model Types** table, the **Implementations** table, the **Software model types and implementations** table, and the **Software model type classifications** table.

The **Software Model Types** table contains the name of each software model type and also the literature sources used to identify it.

The **Implementations** table names each implementation and the vendor that developed it.

The **Software model types and Implementations** table links software model types to implementations. Each row links one software model type to an implementation.

The **Software model type classifications** table contains one row for each software model type used to assess the measurement framework components. Each row contains the actual classification scheme used for a software model type.

Data Collection Instruments

Data collected was documented using the **SMC Test Case Types** table, the **Software model classifications** table, and the **Software model type Test Cases** table.

The **SMC Test Case Types** table is used to categorise the kinds of test cases for software model types. Each row is given a unique identifier (**Test Case Type ID**) and represents the criteria required for to define a test case of the given type.

The **Software model classifications** table contains a number of software models based on a software model type. There is one table for each software model type. Each row represents a classification of a software model using the software model type classification from the software model type classifications table. Each row is given a unique identifier (**SMI ID**) and includes **Model Classification** for a software model based on the software model type.

The **Software model type Test Cases** table is used to document results of classification for different software models of the same software model type. For each software model type with an implementation there is one **Software model type Test Cases** table. There is one row for each test case. Each row includes:

- The **Test Case Type ID** from the **SMC Test Case Types** table to identify the type of test case.
- The **Expected model classification** by citing the **SMI ID** from the **Software model classifications** table.
- The **Actual model classification** by citing the **SMI ID** from the **Software model classifications** table.
- The **Expected Size** for measuring the size of the software model, that is, the size of the **Expected model classification**.
- The **Actual Size** measured for the software model, that is, the size of the **Actual model classification**.

Collection Process Description

1. A series of test case types are defined and documented using the **SMC Test Case Types** table. Each type of test case has a separate row with a **Test Case Type ID**.
2. For each row in the **Software Model Types and Implementations** table a new **Software model type Test Cases** table is defined.
3. The **Actual Classification Scheme** for the software model type from the **Software model type classifications** table are included in the **Software model type Test Cases** table.
4. For each **Test Case Type ID** in the **SMC Test Case Types** table, one or more test cases are defined in each **Software model type Test Cases** table. This included expected results for classification of the software model (**Expected model**

classification) and measurement of its size (**Expected size**). This was done without using the automated version of the measurement framework.

5. Each test case in each **Software model type Test Cases** table is applied to test measurement of size and classification of the software model. Actual results using the automated version of the measurement framework for classification of the software model (**Actual model classification**) and measurement of its size (**Actual size**) are recorded in the same row for the test case.

Variables

Independent Variables

The following independent variables are obtained after execution of the collection process for the software model classification phase.

Independent variables consist of:

1. The **Software model types** table.
2. The **Implementations** table.
3. The **Software model types and implementations** table.
4. The **Software Model Type Classifications** table.
5. The **SMC Test Case Types** table.
6. The **Test Case Type ID, Test Case Description, Expected model classification**, and **Expected Size** in each row of each **Software model type Test Cases** table.
7. The MLC instances of the Software Model Type Classification MLC.
8. The MLC instances of the Software Model Type Set Theory MLC.
9. The MLC instances of the Amount of Reuse Measurement M1 Model Specifier MLC.
10. The Software Model Classification MLC.
11. The Software Model Set Theory MLC.
12. The Amount of Reuse Measurement Model Specifier MLC.

Dependent Variables

The following dependent variables are obtained after execution of the collection process for the software model classification methodology phase.

Dependent variables consist of:

1. The **Actual model classification** and **Actual Size** in each row of each **Software model type Test Cases** table.
2. The MLC instances of the Software Model Classification MLC.
3. The MLC instances of the Software Model Set Theory MLC.
4. The MLC instances of the Amount of Reuse Measurement Model Specifier MLC.

Analysis Process

Hypothesis Measures

Base indicators, coverage indicators, and limitation indicators used in this phase are tabulated in Table 4-12, Table 4-13, and Table 4-14. See Appendix C3 for a guide to interpretation of the hypothesis measures used in this methodology phase.

Table 4-12 lists the base indicators used in the software model classification methodology phase to support calculations for coverage and limitation indicators. Base indicators include the number of software model types and software model type classifications.

Table 4-12: Base Indicators used in Software Model Classification

Model Type Category	Names of Measures
All Software Model Types	**SMT_BI, SMTI_BI, SMTC_BI, SMCTCTR_BI, SMTTC_BI, SMTTCR_BI, SMTTCP_BI**
Object-oriented/UML Software Model Types	**OU_SMT_BI, OU_SMTI_BI, OU_SMTC_BI, OU_SMTTC_BI, OU_SMTTCR_BI, OU_SMTTCP_BI**

Below are four examples of base indicators from the guide to interpretation for this methodology phase (Appendix C3).

Name of Measure	Calculation
SMTC_BI (Software Model Type Classifications Base Indicator)	#rows in the **Software model type classifications** table
SMCTCTR_BI (SMC Test Case Types Base Indicator)	#rows in the **SMC Test Case Types** table
SMTTC_BI (Software Model Type Test Cases Base Indicator)	#number of **Software Model Type Test Cases** tables
SMTTCR_BI (Software Model Test Cases Base Indicator)	#rows in all **Software Model Type Test Cases** tables
SMTTCP_BI (Software Model Test Case Pairs Base Indicator)	#pairs of **Software Model Type Test Cases** $\text{tables}_{1,2}$ where (**SMTI ID**$_1$ ≠ **SMTI ID**$_2$)

SMTC_BI: This indicator is the number of software model type classifications in the **Software model type classifications** table. The indicator represents the number and kind of classifications available for different software model types.

SMCTCTR_BI: This indicator is the number of test case types identified for the purposes of this thesis and located as a number of rows in the **SMC Test Case Types** table.

SMTTC_BI: This indicator is the number of software model types with implementations that were used to test software model classification and is the number of **Software model type Test Cases** tables.

SMTTCR_BI: This indicator is the total number of test cases for all software model types with implementations that were used to test software model classification. The number is the total number of rows in all **Software model type Test Cases** tables.

SMTTCP_BI: This indicator is the number of pairs for all software model types with implementations that were used to test software model classification. The value is obtained by counting all the possible pairs of **Software model type Test Cases** tables with different values for **SMTI ID** for each item in a pair ($^{\#\text{Software model type Test Cases tables}}C_2$, or the sum of all values between zero and SMTTC_BI subtract SMTTC_BI).

Table 4-13 lists the coverage indicators used in the software model classification methodology phase.

Table 4-13: Measurement Specifications for Coverage Indicators

Model Type Category	Names of Measures
All Software Model Types	**Category 1: MCC_CI, SMC_CI, SMCO_CI, SSM_CI, PSMC_CI, PSSM_CI** **Category 2: UMC_CI, USM_CI, PUMC_CI, PUSM_CI**
Object-oriented/UML Software Model Types	**Category 1: OU_MCC_CI, OU_SMC_CI, OU_SMCO_CI, OU_SSM_CI, OU_PSMC_CI, OU_PSSM_CI** **Category 2: OU_UMC_CI, OU_USM_CI, OU_PUMC_CI, OU_PUSM_CI**

Category 1 indicators measure the level of successful classification and measurement of size for different software models.

Category 2 indicators measure the degree of failure at classification and measurement of size for different software models.

Below are four examples of coverage indicators from the guide to interpretation (Appendix C3). The first and second examples are category 1 indicators that quantify outcome 1 from the experiment overview. The second and third examples are category 2 indicators that quantify outcome 2 from the experiment overview.

Name of Measure	Calculation
SMC_CI (Successful Model Classification Coverage Indicator)	#rows in all **Software model type Test Cases** tables where **Expected Model Classification** = **Actual Model Classification**
SSM_CI (Successful Size Measurement Coverage Indicator)	#rows in all **Software model type Test Cases** tables where **Expected Size** = **Actual Size**
UMC_CI (Unsuccessful Model Classification Coverage Indicator)	#rows in all **Software model type Test Cases** tables where **Actual Model Classification** = "UNABLE TO CLASSIFY SOFTWARE MODEL"
USM_CI (Unsuccessful Size Measurement Coverage Indicator)	#rows in all **Software model type Test Cases** tables where **Expected Size** ≠ **Actual Size**

SMC_CI: This indicator attempts to express the degree of successful classification relative to the number of classification attempts. The value is obtained by counting the number successful classification attempts of software models for all software

model types. This value is compared with SMTTCR_BI. A high value indicates that the degree of successful classification is high. The highest possible value for SMC_CI is SMTTCR_BI. A low value indicates that the degree of successful classification is low. The lowest possible value for SMC_CI is zero.

SSM_CI: This indicator attempts to express the degree of successful size measurement relative to the number of size measurement attempts. The value is obtained by counting the number of successful size measurement attempts for all software model types. This value is compared with SMTTCR_BI. A high value indicates that the degree of successful size measurement is high. The highest possible value for SSM_CI is SMTTCR_BI. A low value indicates that the degree of successful size measurement is low. The lowest possible value for SSM_CI is zero.

UMC_CI: This indicator attempts to express the degree of unsuccessful classification relative to the number of classification attempts. The value is obtained by counting the number unsuccessful classification attempts of software models for all software model types. This value is compared with SMTTCR_BI. A high value indicates that the degree of unsuccessful classification is high. The highest possible value for UMC_CI is SMTTCR_BI. A low value indicates that the degree of unsuccessful classification is low. The lowest possible value for UMC_CI is zero.

USM_CI: This indicator attempts to express the degree of unsuccessful size measurement relative to the number of size measurement attempts. The value is obtained by counting the number unsuccessful size measurement attempts for all software model types. This value is compared with SMTTCR_BI. A high value indicates that the degree of unsuccessful size measurement is high. The highest possible value for USM_CI is SMTTCR_BI. A low value indicates that the degree of unsuccessful size measurement is low. The lowest possible value for USM_CI is zero.

Table 4-14 lists the limitation indicators used in the software model classification methodology phase.

Table 4-14: Limitation Indicators used in Software Model Classification

Model Type Category	Names of Measures
All Software Model Types	**Category 1: PCSMTI_LI, CSMTI_LI** **Category 2: SMSCST_LI, SMSCSM_LI, SMUCST_LI, SMUCSM_LI, SMUCUT_LI, SMUCUM_LI, SMSCUT_LI, SMSCUM_LI** **Category 3: SSTCCTC_LI, SCTCCTC_LI, SSTCDTC_LI, SCTCDTC_LI**
Object-oriented/UML Software Model Types	**Category 1: OU_PCSMTI_LI, OU_CSMTI_LI** **Category 2: OU_SMSCST_LI, OU_SMUCST_LI, OU_SMSCSM_LI, OU_SMUCSM_LI, OU_SMUCUT_LI, OU_SMSCUT_LI, OU_SMUCUM_LI, OU_SMSCUM_LI** **Category 3: OU_SSTCCTC_LI, OU_SCTCCTC_LI, OU_SSTCDTC_LI, OU_SCTCDTC_LI**

Category 1 indicators measure coverage of software model types available relative to the number of attempts at classification of software model types.

Category 2 indicators measure the degree of successful classification and measurement of size for different software models relative to the results for classification of software model types.

Category 3 indicators measure the degree of successful classification and measurement of size for different software model types relative to the similarity between software mode type classifications for different software model types.

Below are five examples of limitation indicators taken from the guide to interpretation. The first example is a category 1 indicator that quantifies outcome 3 from the experiment overview. The second and third examples are category 2 indicators that quantify outcome 4 from the experiment overview. The fourth example is a category 3 indicator that quantifies outcome 5 from the experiment overview. The fifth example is a category 3 indicator that quantifies outcome 6 from the experiment overview.

Name of Measure	Calculation
CSMTI_LI (Coverage of Software Model Types with Implementations Limitation Indicator)	#rows in the **Software model type classifications** table where **SMTI ID** = **SMTI ID** in one row in the **Software model types and Implementations** table
SMSCST_LI (Software Model Successful Classification Successful Test Case Limitation Indicator)	#number of **Software model type Test Cases** tables where (**Actual Classification Scheme** ≠ "NOT ABLE TO CLASSIFY SOFTWARE MODEL TYPE") × (#rows in **Software model type Test Cases** table where **Expected Model Classification** = **Actual Model Classification**)
SMUCST_LI (Software Model Unsuccessful Classification Successful Test Case Limitation Indicator)	#number of **Software model type Test Cases** tables where (**Actual Classification Scheme** = "NOT ABLE TO CLASSIFY SOFTWARE MODEL TYPE") × (#rows in **Software model type Test Cases** table where **Expected Model Classification** = **Actual Model Classification**)
SSTCCTC_LI (Successful Size Test Case Common Type Classification Limitation Indicator)	#pairs of **Software model type Test Cases** $tables_{1,2}$ where (**SMTI ID**$_1$ ≠ **SMTI ID**$_2$) and (**Actual Classification Scheme**$_1$ = **Actual Classification Scheme**$_2$) and (**Actual Classification Scheme**$_1$ ≠ "NOT ABLE TO CLASSIFY SOFTWARE MODEL TYPE") and (**Actual Classification Scheme**$_2$ ≠ "NOT ABLE TO CLASSIFY SOFTWARE MODEL TYPE") and (#rows in **Software model type Test Cases** $table_1$ where **Expected Size** = **Actual Size** ≥ 1) and (#rows in **Software model type Test Cases** $table_2$ where **Expected Size** = **Actual Size** ≥ 1)
SSTCDTC_LI (Successful Size Test Case Different Type Classification Limitation Indicator)	#pairs of **Software model type Test Cases** $tables_{1,2}$ where (**SMTI ID**$_1$ ≠ **SMTI ID**$_2$) and (**Actual Classification Scheme**$_1$ ≠ **Actual Classification Scheme**$_2$) and (**Actual Classification Scheme**$_1$ ≠ "NOT ABLE TO CLASSIFY SOFTWARE MODEL TYPE") and (**Actual Classification Scheme**$_2$ ≠ "NOT ABLE TO CLASSIFY SOFTWARE MODEL TYPE") and (#rows in **Software model type Test Cases** $table_1$ where **Expected Size** = **Actual Size** ≥ 1) and (#rows in **Software model type Test Cases** $table_2$ where **Expected Size** = **Actual Size** ≥ 1)

CSMTI_LI: This indicator attempts to assesses the number of classification attempts relative to the number of software model types with implementations. The value is calculated by counting the number of software model types and implementations where a classification attempt is made. The value is compared with SMTI_BI. A high

value indicates many classification attempts made relative to software model types with implementations. The highest possible value for CSMTI_LI is equal to SMTI_BI. A low value indicates few classification attempts made relative to software model types with implementations. The lowest possible value for CSMTI_LI is equal to zero.

SMSCST_LI: This indicator attempts to assess the degree of successful classification for software model types with successful software model type classifications. The value is obtained by counting the number of successful software model classifications for all Software model types that have a successful software model type classification. This value is compared to SMTTCR_BI. A high value indicates a high degree of successful software model classification for software model types with successful software model type classifications. The highest possible value for SMSCST_LI is SMTTCR_BI. A low value indicates a low degree of successful software model classification for software model types with successful software model type classifications. The lowest possible value for SMSCST_LI is zero.

SMUCST_LI: This indicator attempts to assess the degree of successful classification for software model types with unsuccessful software model type classifications. The value is obtained by counting the number of successful software model classifications for all Software model types that have an unsuccessful software model type classification. This value is compared to SMTTCR_BI. A high value indicates a high degree of successful software model classification for software model types with unsuccessful software model type classifications. The highest possible value for SMUCST_LI is SMTTCR_BI. A low value indicates a low degree of successful software model classification for software model types with unsuccessful software model type classifications. The lowest possible value for SMUCST_LI is zero.

SSTCCTC_LI: This indicator attempts to assess the degree of common software model type classification for different software model types with successful software model type classifications and at least one successful attempt at size measurement. The value is obtained by counting the total number of software model type pairs that are different in name but have the same successful software model type classification and at least one successful attempt at size measurement. The value is compared to SMTTCP_BI. A high value indicates a high degree of common software model type classification. The highest possible value for SSTCCTC_LI is equal to SMTTCP_BI. A low value indicates a low degree of common software model type classification. The lowest possible value for SSTCCTC_LI is zero.

SSTCDTC_LI: This indicator attempts to assess the degree of different software model type classification for different software model types with successful software model type classifications and at least one successful attempt at size measurement. The value is obtained by counting the total number of software model type pairs that are different in name and have a different successful software model type classification and at least one successful attempt at size measurement. The value is compared to SMTTCP_BI. A high value indicates a high degree of different software model type classification. The highest possible value for SSTCDTC_LI is equal to SMTTCP_BI. A low value indicates a low degree of different software model type classification. The lowest possible value for SSTCDTC_LI is zero.

Assessment Criteria

Use of the hypothesis measures to test the hypotheses is illustrated in:

- Table 4-15 for **H0a** and **H0b**.
- Table 4-16 for **H1** via **SH 1.2**, **SH 1.3**, and **SH 1.4**.
- Table 4-17 for **H2** via **SH 2.2**, **SH 2.3**, and **SH 2.4**.

Refer to Appendix C3 for further details on descriptions for assessment criteria.

Table 4-15 names the hypothesis measures used for testing the null hypothesis **H0a** and **H0b**.

Table 4-15: Assessment of **H0a/b** using Hypothesis measures

Hypothesis Part	Name of Measure
H0a	PSMC_CI, PUMC_CI, PSSM_CI, PUSM_CI OU_PSMC_CI, OU_PUMC_CI, OU_PSSM_CI, OU_PUSM_CI
H0b	PCSMTI_LI, CSMTI_LI SMSCST_LI, SMUCST_LI, SMSCSM_LI, SMUCSM_LI SMUCUT_LI, SMSCUT_LI, SMUCUM_LI, SMSCUM_LI SSTCCTC_LI, SCTCCTC_LI SSTCDTC_LI, SCTCDTC_LI OU_PCSMTI_LI, OU_CSMTI_LI OU_SMSCST_LI, OU_SMUCST_LI, OU_SMSCSM_LI, OU_SMUCSM_LI OU_SMUCUT_LI, OU_SMSCUT_LI, OU_SMUCUM_LI, OU_SMSCUM_LI OU_SSTCCTC_LI, OU_SCTCCTC_LI OU_SSTCDTC_LI, OU_SCTCDTC_LI

How the values for the hypothesis measures support H0 is defined in the guide to interpretation (Appendix C3). Below are some examples from the guide to interpretation that illustrate this.

The truth of **H0a** is *supported* if:

- There must be no successful attempts at classification of software models (PSMC_CI = 0) and all classification attempts of software models must be unsuccessful (PUMC_CI = 100), and
- There must be no successful attempts at measuring the size of software models (PSSM_CI = 0) and all attempts at measuring size of software models must be unsuccessful (PUSM_CI = 100).

To *deny* **H0a**:

- There must be some successful attempts at classification of software models (PSMC_CI > 0) and not all classification attempts of software models must be unsuccessful (PUMC_CI < 100).
- There must be some successful attempts at measuring the size of software models (PSSM_CI > 0) and not all attempts at measuring size of software models must be unsuccessful (PUSM_CI < 100).

The truth of **H0b** is *supported* if:

- All software model types with an implementation must have at least one attempt at classification of the software model type (CSMTI_LI = SMTI_BI) and there must be at least one software model type with an implementation included in the study (SMTI_BI >= 1).
- All test cases must have successful software model classifications for all software model types with successful software model type classifications (SMSCST_LI = SMTTCR_BI).
- All test cases must have successful software model classifications for all software model types with unsuccessful software model type classifications (SMUCST_LI = SMTTCR_BI).
- All software model types with at least one successful attempt at size measurement used a common software model type classification scheme (SSTCCTC_LI = SMTTCP_BI) with at least two software model types included in the study (SMTTC_BI > 1).
- No software model types with at least one successful attempt at size measurement used a different software model type classification scheme (SSTCDTC_LI = 0) with at least two software model types included in the study (SMTTC_BI > 1).

To *deny* **H0b**:

- Some software model types with an implementation do not have one single attempt at classification of the software model type (CSMTI_LI < SMTI_BI) and there must be at least one software model type with an implementation included in the study (SMTI_BI >= 1).
- Not all test cases have successful software model classifications for all software model types with successful software model type classifications (SMSCST_LI < SMTTCR_BI).
- Some test cases do not have successful software model classifications for all software model types with unsuccessful software model type classifications (SMUCST_LI < SMTTCR_BI).
- Not all software model types with at least one successful attempt at size measurement used a common software model type classification scheme (SSTCCTC_LI < SMTTCP_BI) with at least two software model types included in the study (SMTTC_BI > 1).
- Some software model types with at least one successful attempt at size measurement used a different software model type classification scheme (SSTCDTC_LI > 0) with at least two software model types included in the study (SMTTC_BI > 1).

Table 4-16 names the hypothesis measures used for testing hypothesis **H1** via **SH 1.2**, **SH 1.3**, and **SH 1.4**.

Table 4-16: Assessment of Hypothesis **H1** using hypothesis measures and **SH 1.2**, **SH 1.3**, and **SH 1.4**.

Sub-Hypothesis	Measures Used
SH 1.2	MCC_CI, SMC_CI, UMC_CI, PSMC_CI, PUMC_CI, SMTTC_BI OU_MCC_CI, OU_SMC_CI, OU_UMC_CI, OU_PSMC_CI, OU_PUMC_CI, OU_SMTTC_BI
SH 1.3	SMCO_CI, SSM_CI, USM_CI, PSMS_CI, PUMS_CI, SMTTC_BI OU_SMCO_CI, OU_SSM_CI, OU_USM_CI, OU_PSMS_CI, OU_PUMS_CI, OU_SMTTC_BI
SH 1.4	SMCO_CI, SSM_CI, USM_CI, PSMS_CI, PUMS_CI, SMTTCR_BI OU_SMCO_CI, OU_SSM_CI, OU_USM_CI, OU_PSMS_CI, OU_PUMS_CI, OU_SMTTCR_BI

How the values for the hypothesis measures support H1 are detailed in the guide to interpretation (Appendix C3). Below are some examples from the guide to interpretation that illustrate this.

Sub-Hypothesis	Name of Measure	Value to Support Hypothesis	Value to Not Support Hypothesis
SH 1.2	**SMC_CI SMTTC_BI**	SMC_CI > 1 and SMTTC_BI > 1 and SMTTCR_BI > 1	SMC_CI < 2 and SMTTC_BI > 1 and SMTTCR_BI > 1
	UMC_CI, SMTTC_BI	UMC_CI + 1 < SMTTCR_BI and SMTTC_BI > 1 and SMTTCR_BI > 1	UMC_CI + 1> = SMTTCR_BI and SMTTC_BI > 1 and SMTTCR_BI > 1
SH 1.3	**SSM_CI SMTTC_BI**	SSM_CI > 1 and SMTTC_BI > 1	SSM_CI < 2 and SMTTC_BI > 1
	USM_CI, SMTTC_BI	USM_CI + 1 < SMTTCR_BI and SMTTC_BI > 1	USM_CI + 1 >= SMTTCR_BI and SMTTC_BI > 1
SH 1.4	**SSM_CI SMTTCR_BI**	SSM_CI > 1 and SMTTCR_BI > 1	SSM_CI < 2 and SMTTCR_BI > 1
	USM_CI, SMTTCR_BI	USM_CI + 1 < SMTTCR_BI and SMTTCR_BI > 1	USM_CI + 1 >= SMTTCR_BI and SMTTCR_BI > 1

The truth of **H1** is *supported* if the truth of **SH 1.2**, **SH 1.3**, or **SH 1.4** is *supported.*

To *support* **SH 1.2**:

- There must be at least two successful classification attempts of software models (SMC_CI > 1) with at least two software model types tested (SMTTC_BI > 1) and at least two attempts at classification of a software model (SMTTCR_BI > 1).
- The number of unsuccessful classification attempts of software models must be at least two less than the number of attempts at classification of software models (UMC_CI + 1 < SMTTCR_BI) with at least two software model types tested (SMTTC_BI > 1) and at least two attempts at classification of a software model (SMTTCR_BI > 1).

SH 1.2 is *not supported* if:

- There is only one successful attempt at classification of software models (SMC_CI < 2) with at least two software model types tested (SMTTC_BI > 1) and at least two attempts at classification of a software model (SMTTCR_BI > 1).
- The number of unsuccessful attempts at classification of software models plus one is equal to or more than the number of classification attempts of software models (UMC_CI + 1 >= SMTTCR_CI) with at least two software model types tested (SMTTC_BI > 1) and at least two attempts at classification of a software model (SMTTCR_BI > 1).

To *support* **SH 1.3**:

- There must be at least two successful attempts at measurement of size (SSM_CI > 1) with at least two software model types tested (SMTTC_BI > 1).
- The number of unsuccessful attempts at measurement of size must be at least two less than the number of attempts at measurement of size (USM_CI + 1 < SMTTCR_BI) with at least two software model types tested (SMTTC_BI > 1).

SH 1.3 is *not supported* if:

- There is only one successful attempt at measurement of size (SSM_CI < 2) with at least two software model types tested (SMTTC_BI > 1).
- The number of unsuccessful attempts at measurement of size plus one is equal to or more than the number of attempts at measurement of size (USM_CI + 1 >= SMTTCR_CI) with at least two software model types tested (SMTTC_BI > 1).

To *support* **SH 1.4**:

- There must be at least two successful attempts at measurement of size (SSM_CI > 1) with at least two attempts at measurement of size (SMTTCR_BI > 1).
- The number of unsuccessful attempts at measurement of size must be at least two less than the number of attempts at measurement of size (USM_CI + 1 < SMTTCR_BI) with at least two attempts at measurement of size (SMTTCR_BI > 1).

SH 1.4 is *not supported* if:

- There is only one successful attempt at measurement of size (SSM_CI < 2) with at least two attempts at measurement of size (SMTTCR_BI > 1).
- The number of unsuccessful attempts at measurement of size plus one is equal to or more than the number of attempts at measurement of size (USM_CI + 1 >= SMTTCR_CI) with at least two attempts at measurement of size (SMTTCR_BI > 1).

Table 4-17 names the hypothesis measures used for testing hypothesis **H2** via **SH 2.2**, **SH 2.3**, and **SH 2.4**.

Table 4-17: Assessment of Hypothesis **H2** using Hypothesis measures and **SH 2.2**, **SH 2.3**, and **SH 2.4**.

Sub-Hypothesis	Measures Used
SH 2.2	**PCSMT_LI, CSMTI_LI** **SMSCST_LI, SMUCST_LI, SMUCUT_LI, SMSCUT_LI** **SCTCCTC_LI, SCTCDTC_LI, SMTTC_BI** **OU_PCSMT_LI, OU_CSMTI_LI, OU_SMSCST_LI, OU_SMUCST_LI,** **OU_SMUCUT_LI, OU_SMSCUT_LI, OU_SCTCCTC_LI, OU_SCTCDTC_LI,** **OU_SMTTC_BI**
SH 2.3	**SMSCSM_LI, SMUCSM_LI, SMUCUM_LI, SMSCUM_LI** **SSTCCTC_LI, SSTCDTC_LI, SMTTC_BI** **OU_SMSCSM_LI, OU_SMUCSM_LI, OU_SMUCUM_LI, OU_SMSCUM_LI,** **OU_SSTCCTC_LI, OU_SSTCDTC_LI, OU_SMTTC_BI**
SH 2.4	**SMSCSM_LI, SMUCSM_LI, SMUCUM_LI, SMSCUM_LI, SMTTCR_BI** **OU_SMSCSM_LI, OU_SMUCSM_LI, OU_SMUCUM_LI, OU_SMSCUM_LI,** **OU_SMTTCR_BI**

How the values for the hypotheses support H2 are detailed in the guide to interpretation (Appendix C3). Below are some examples from the guide to interpretation that illustrate this.

Sub-Hypothesis	Name of Measure	Value to Support Hypothesis	Value to Not Support Hypothesis
SH 2.2	**CSMTI_LI, SMTI_BI**	CSMTI_LI < SMTI_BI and SMTI_BI > 0	CSMTI_LI = SMTI_BI and SMTI_BI > 0
	SMSCST_LI, SMTTC_BI	SMSCST_LI < SMTTCR_BI and SMTTC_BI > 1	SMSCST_LI = SMTTCR_BI and SMTTC_BI > 1
	SMUCST_LI, SMTTC_BI	SMUCST_LI < SMTTCR_BI and SMTTC_BI > 1	SMUCST_LI = SMTTCR_BI and SMTTC_BI > 1
SH 2.3	**SMSCSM_LI, SMTTC_BI**	SMSCSM_LI < SMTTCR_BI and SMTTC_BI > 1	SMSCSM_LI = SMTTCR_BI and SMTTC_BI > 1
	SMUCSM_LI, SMTTC_BI	SMUCSM_LI < SMTTCR_BI and SMTTC_BI > 1	SMUCSM_LI = SMTTCR_BI and SMTTC_BI > 1
	SSTCCTC_LI, SMTTC_BI	SSTCCTC_LI < SMTTCP_BI and SMTTC_BI > 1	SSTCCTC_LI = SMTTCP_BI and SMTTC_BI > 1
	SSTCDTC_LI, SMTTC_BI	SSTCDTC_LI >0 and SMTTC_BI > 1	SSTCDTC_LI = 0 and SMTTC_BI > 1
SH 2.4	**SMSCSM_LI, SMTTCR_BI**	SMSCSM_LI < SMTTCR_BI and SMTTCR_BI > 1	SMSCSM_LI = SMTTCR_BI and SMTTCR_BI > 1
	SMUCSM_LI, SMTTCR_BI	SMUCSM_LI < SMTTCR_BI and SMTTCR_BI > 1	SMUCSM_LI = SMTTCR_BI and SMTTCR_BI > 1

The truth of **H2** is *supported* if the truth of **SH 2.2**, **SH 2.3**, or **SH 2.4** is *supported.*

To *support* **SH 2.2**:

- Some software model types with an implementation do not have one single attempt at classification of the software model type (CSMTI_LI < SMTI_BI) and there must be at least one software model type with an implementation included in the study (SMTI_BI > 0).
- Not all test cases have successful software model classifications for all software model types with successful software model type classifications (SMSCST_LI < SMTTCR_BI) with at least two software model types included in the study (SMTTCR_BI > 1).
- Some test cases do not have successful software model classifications for all software model types with unsuccessful software model type classifications (SMUCST_LI < SMTTCR_BI) with at least two software model types included in the study (SMTTCR_BI > 1).

SH 2.2 is *not supported* if:

- All software model types with an implementation must have at least one attempt at classification of the software model type (CSMTI_LI = SMTI_BI) and there must be at least one software model type with an implementation included in the study (SMTI_BI > 0).
- All test cases must have successful software model classifications for all software model types with successful software model type classifications (SMSCST_LI = SMTTCR_BI) with at least two software model types included in the study (SMTTCR_BI > 1).
- All test cases must have successful software model classifications for all software model types with unsuccessful software model type classifications (SMUCST_LI = SMTTCR_BI) with at least two software model types included in the study (SMTTCR_BI > 1).

To *support* **SH 2.3**:

- Not all test cases have successful attempts at size measurement for all software model types with successful software model type classifications (SMSCSM_LI < SMTTCR_BI) with at least two software model types included in the study (SMTTC_BI > 1).
- Not all test cases have successful attempts at size measurement for all software model types with unsuccessful software model type classifications (SMUCSM_LI < SMTTCR_BI) with at least two software model types included in the study (SMTTC_BI > 1).

- Not all software model types with at least one successful attempt at size measurement used a common software model type classification scheme (SSTCCTC_LI < SMTTCP_BI) with at least two software model types included in the study (SMTTC_BI > 1).
- Some software model types with at least one successful attempt at size measurement used a different software model type classification scheme (SSTCDTC_LI > 0) with at least two software model types included in the study (SMTTC_BI > 1).

SH 2.3 is *not supported* if:

- All test cases must have successful attempts at size measurement for all software model types with successful software model type classifications (SMSCSM_LI = SMTTCR_BI) with at least two software model types included in the study (SMTTC_BI > 1).
- All test cases must have successful attempts at size measurement for all software model types with unsuccessful software model type classifications (SMUCSM_LI = SMTTCR_BI) with at least two software model types included in the study (SMTTC_BI > 1).
- All software model types with at least one successful attempt at size measurement used a common software model type classification scheme (SSTCCTC_LI = SMTTCP_BI) with at least two software model types included in the study (SMTTC_BI > 1).
- No software model types with at least one successful attempt at size measurement used a different software model type classification scheme (SSTCDTC_LI = 0) with at least two software model types included in the study (SMTTC_BI > 1).

To *support* **SH 2.4**:

- Not all test cases have successful attempts at size measurement for all software model types with successful software model type classifications (SMSCSM_LI < SMTTCR_BI) with at least two attempts at size measurement in the study (SMTTCR_BI > 1).
- Not all test cases have successful attempts at size measurement for all software model types with unsuccessful software model type classifications (SMUCSM_LI < SMTTCR_BI) with at least two attempts at size measurement in the study (SMTTCR_BI > 1).

SH 2.4 is *not supported* if:

- All test cases must have successful attempts at size measurement for all software model types with successful software model type classifications (SMSCSM_LI = SMTTCR_BI) with at least two attempts at size measurement in the study (SMTTCR_BI > 1).
- All test cases must have successful attempts at size measurement for all software model types with unsuccessful software model type classifications (SMUCSM_LI = SMTTCR_BI) with at least two attempts at size measurement in the study (SMTTCR_BI > 1).

To summarise, **H0a/b** is *supported* if:

- all attempts at classification and measurement of size with different software models fail (**H0a**), and
- all available software model types are classified (**H0b**), and
- all attempts at classification and measurement of size for different software models are successful regardless of the results of classification for software model types (**H0b**), and
- all successful attempts at classification and measurement of size for different software model types use the same software model type classification (**H0b**), and

H0a/b is *denied* if:

- at least one attempt at classification or measurement of size with different software models is successful (**H0a**), or
- not all available software model types are classified (**H0b**), or
- at least one attempt at classification or measurement of size for different software models is successful because of the results of classification for software model types (**H0b**), or
- at least two successful attempts at classification or measurement of size for different software model types use a different software model type classification (**H0b**).

H1 is *supported* if:

- Some attempts at classification of different software models are successful for different software model types (SH 1.2).
- Some attempts at measurement of size are successful for different software model types (SH 1.3).
- Some attempts at measurement of size are successful for different software models (SH 1.4).

H1 is *not supported* if:

- Only one attempt at classification of different software models is successful for one software model type (SH 1.2).
- Only one attempt at measurement of size is successful for one software model type (SH 1.3).
- Only one attempt at measurement of size is successful for different software models (SH 1.4).

H2 is *supported* if:

- Not all software model types available are classified (SH 2.2).
- Some attempts at classification of different software models are successful because of results for classification of different software model types (SH 2.2).
- Successful attempts at classification of different software models use different software model type classifications (SH 2.2).
- Some attempts at measurement of size with different software models are successful because of results for classification of different software model types (SH 2.3).
- Successful attempts at measurement of size with different software models use different software model type classifications (SH 2.3).
- Some attempts at measurement of size with different software models are successful because of results for classification of any software model type. (SH 2.4).

H2 is *not supported* if:

- All software model types available are classified (SH 2.2).
- Only one attempt at classification of different software models is successful because of results for classification of different software model types (SH 2.2).
- Successful attempts at classification of different software models use the same software model type classification (SH 2.2).
- Only one attempt at measurement of size with different software models is successful because of results for classification of different software model types (SH 2.3).
- Successful attempts at measurement of size with different software models use the same software model type classification (SH 2.3).
- Only one attempt at measurement of size with different software models is successful because of results for classification of any software model type (SH 2.4).

This concludes the assessment criteria in the software model classification methodology phase for hypotheses **H0a**, **H0b**, **H1/SH 1.2 - 1.4**, and **H2/SH 2.3 - 2.4**.

Analysis Report Input

The methodology phase analysis reports used in this phase were:

The Software Model Classification Analysis Report. See the section under the heading "Software Model Classification Report" in Appendix D2 for a complete example.

The hypothesis analysis reports used in this phase are:

The H0 Analysis Report. See the section under the heading "H0 Hypothesis Analysis Report" in Appendix D5 for a complete example.

The H1 Analysis Report. See the section under the heading "H1 Hypothesis Analysis Report" in Appendices D6 – D7 for a complete example.

The H2 Analysis Report. See the section under the heading "H2 Hypothesis Analysis Report" in Appendices D8 – D9 for a complete example.

Analysis Report Output

The analysis reports generated by this methodology phase are:

The Software Model Classification Analysis Report. This report is generated by adding all measures for independent and dependent variables in the software model classification methodology phase along with the values obtained. In addition, values for all hypothesis measures are added to the report.

The analysis reports modified in this methodology phase were:

The H0 Analysis Report.

The H1 Analysis Report.

The H2 Analysis Report.

The above hypothesis analysis reports include the values obtained for all hypothesis measures from the software model classification methodology phase. Identification of the level of support for each of the hypotheses is also included and relevant totals for sub-hypotheses and the hypothesis are updated.

Analysis Process Description

1. Calculate the values of the measures derived from the independent variables and dependent variables for the software model classification methodology phase.
2. Enter the names and values for these measures in the Software Model Classification Methodology Phase Analysis Report.
3. Calculate the values of the base indicators, coverage indicators, and limitation indicators for hypothesis measures listed under hypothesis measures for the software model classification methodology phase.
4. Enter the values of the hypothesis measures in the appropriate rows of the Software Model Classification Methodology Phase Analysis report.
5. Copy the values for the hypothesis measures in the Software Model Classification Methodology Phase Analysis Report into the appropriate rows in the hypothesis analysis reports. For each of theses rows enter the MP_ID in the actual value column as "SMC" and add the required values to support and deny the hypothesis in their respective columns.
6. Update the sub-totals and totals in the **H0** hypothesis analysis report, the **H1** hypothesis analysis report, the **H2** hypothesis analysis report, the **H3** hypothesis analysis report, the **H4** hypothesis analysis report, and the Software Model Classification Analysis Report.

Measurement Testing: MT

Experiment Overview

In section 4.3 the third claim made was that the measurement framework can use software model type classifications to measure the amount of reuse of different software models in a predictable and consistent way. This methodology phase is an experiment designed to test if this is true. The experiment can be summarised as follows:

Step 1. All software model types from the software model type classification methodology phase are included in this methodology phase. This includes software model types implemented on CASE tools and software model types not implemented.

Step 2. For each software model type classified in the software model type classification methodology phase, pairs of software model classifications are made to test various possibilities of reuse measurement based on set theory and the reuse approach.

Step 3. For each pair of software models in Step 2, an attempt is made to measure the amount of reuse of the software model using the measurement framework. This includes each kind of reuse approach. To determine success or failure the actual values for the amount reused using the automated version of the measurement framework are compared to the expected values specified for the software models.

This is the essence of the collection process.

There are five basic outcomes for any given software model type used in the experiment. These outcomes are not all mutually exclusive.

Outcome 1. The software model type was successfully classified and it was possible to measure the amount of reuse for different software models based on the software model type using the measurement framework and the software model type classification. This supports hypothesis H1.

Outcome 2. The software model type was successfully classified and it was not possible to measure the amount of reuse for different software models based on the software model type using the measurement framework and the software model type classification. This does not support hypothesis H1.

Outcome 3. The software model type was never classified and it was not possible to determine if the measurement framework could measure the amount of reuse for different software models based on the software model type. This supports hypothesis H2.

Outcome 4. Successful measurement of the amount of reuse between software models is dependent upon successful classification of their software model type. This supports hypothesis H2.

Outcome 5. The same software model type classification was used to classify different software model types used in the measurement testing methodology phase. This does not support hypothesis H2.

Outcome 6. Different software model type classifications were used to classify different software model types used in the measurement testing methodology phase. This supports hypothesis H2.

Based on the above outcomes, support for hypotheses H1 and H2 can be quantified as follows:

- H1 is supported by the number of software model type classifications used to successfully measure the amount of reuse for different software models.
- H1 is not supported by the number of software model type classifications used to unsuccessfully measure the amount of reuse for different software models.
- H2 is supported by the number of software model types not classified and therefore not used in the measurement testing methodology phase.
- H2 is supported by the number of times a successful measurement of the amount of reuse between two software models is associated with a successful classification of their software model type.

- H2 is supported by the number of software model types with different software model type classifications used to measure the amount of reuse for different software models.
- H2 is not supported by the number of software model types with common software model type classifications used to measure the amount of reuse for different software models.

The six basic quantities above are the essence of hypothesis measures and assessment criteria in this methodology phase. Now follows a description of the methodology phase in more detail.

Collection Process

Assessed Components

The assessed components in the framework for this phase are:

1. The Amount of Reuse Measurement M1 Model Specifier MLC.
2. The Software Model Classification MLC.
3. The Software Model Set Theory MLC.
4. The Amount of Reuse Measurement Model Specifier MLC.

Hypotheses Tested

The hypotheses tested are:

H0a.

H0b.

H1 via **SH 1.5**, **SH 1.6**, **SH 1.7**, **SH 1.8**, **SH 1.9**, **SH 1.10**, **SH 1.11**, and **SH 1.12**.

H2 via **SH 2.5**, **SH 2.6**, **SH 2.7**, **SH 2.8**, **SH 2.9**, **SH 2.10**, **SH 2.11**, and **SH 2.12**.

H0a: A framework based on meta-modelling cannot support measurement of the amount of reuse for any kind of software model.

H0b: A framework based on meta-modelling does not have any limitations in measurement of the amount of reuse with different kinds of software models.

H0: A framework based on meta-modelling that supports measurement of the amount of reuse for different kinds of software models does have advantages.

SH 1.5: A framework based on meta-modelling that can measure the amount of reuse for internal composition reuse for different kinds of software models does have advantages.

SH 1.6: A framework based on meta-modelling that can measure the amount of reuse based on external composition reuse for different kinds of software models does have advantages.

SH 1.7: A framework based on meta-modelling that can measure the amount of reuse for internal generation reuse for different kinds of software models does have advantages.

SH 1.8: A framework based on meta-modelling that can measure the amount of reuse for external generation reuse for different kinds of software models does have advantages.

SH 1.9: A framework based on meta-modelling that can measure the amount of reuse for internal composition reuse for different software models does have advantages.

SH 1.10: A framework based on meta-modelling that can measure the amount of reuse for external composition reuse for different software models does have advantages.

SH 1.11: A framework based on meta-modelling that can measure the amount of reuse for internal generation reuse for different software models does have advantages.

SH 1.12: A framework based on meta-modelling that can measure the amount of reuse for external generation reuse for different software models does have advantages.

H1: A framework based on meta-modelling has limitations in measurement of the amount of reuse with different kinds of software models.

SH 2.5: A framework based on meta-modelling that can measure the amount of reuse for internal composition reuse for different kinds of software models has limitations.

SH 2.6: A framework based on meta-modelling that can measure the amount of reuse based on external composition reuse for different kinds of software models has limitations.

SH 2.7: A framework based on meta-modelling that can measure the amount of reuse for internal generation reuse for different kinds of software models has limitations.

SH 2.8: A framework based on meta-modelling that can measure the amount of reuse for external generation reuse for different kinds of software models has limitations.

SH 2.9: A framework based on meta-modelling that can measure the amount of reuse for internal composition reuse for different software models has limitations.

SH 2.10: A framework based on meta-modelling that can measure the amount of reuse for external composition reuse for different software models has limitations.

SH 2.11: A framework based on meta-modelling that can measure the amount of reuse for internal generation reuse for different software models has limitations.

SH 2.12: A framework based on meta-modelling that can measure the amount of reuse for external generation reuse for different software models has limitations.

Data Specifications

Data specifications used in this methodology phase were the **Software Model Types** table, the **Implementations** table, the **Software model types and implementations** table, and the **Software model type classifications** table.

The **Software Model Types** table contains the name of each software model type and also the literature sources used to identify it.

The **Implementations** table names each implementation and the vendor that developed it. Each implementation is given a unique **Implementation name**.

The **Software model types and Implementations** table links software model types to implementations. Each row links one software model type to an implementation.

The **Software model type classifications** table contains one row for each software model type used to assess the framework components. Each row contains the actual classification scheme used for a software model type.

Data Collection Instruments

Data collected is documented using the **Software model classifications** table, the **MT Test Case Types** table, and the **MT Test Cases** table.

The **Software model classifications** table contains a number of software models based on a software model type. There is one table for each software model type. Each row represents a classification of a software model using the software model type classification from the software model type classifications table. Each row is given a unique identifier (**SMI ID**) and includes **Model Classification** for a software model based on the software model type.

The **MT Test Case Types** table contains the criteria for different kinds of test cases. There is one table for each **Reuse Approach** (either External Composition Reuse, External Generation Reuse, Internal Composition Reuse, or Internal Generation Reuse). Each row in the table is given a unique identifier (**MT Test Case Type ID**) and documents the constraints required to satisfy the kind of test case.

The **MT Test Cases** table documents the results of test cases based on the reuse approach and software model type used. There is one table for each combination of software model type with an implementation and **Reuse Approach**. Each row in the table represents the results of one test case. Each row includes:

- The **MT Test Case Type ID** from the **MT Test Case Types** table to identify the type of test case.
- The **Software Models Used** by citing **SMI IDs** from the **Software model classifications** table. If the **Reuse Approach** includes Composition then **SMI IDs** are given for the Library Model and Application Model. If the **Reuse Approach** includes Generation then **SMI IDs** are given for the Generated Model, and Complete Model.
- The **Expected Results** for the measures or the amount of reuse based on the **Software Models Used**.
- The **Actual Results** obtained for the measures of the amount of reuse with the **Software Models Used**.

Collection Process Description

1. One **MT Test Case Types** table was created for each **Reuse Approach**. A number of test case types were defined in each table.
2. For each combination of an approach to reuse with a software model type linked to an implementation in the **Software model types and Implementations** table one **MT Test Cases** table was created to document results of measurement testing for the software model type based on one approach to reuse.
3. For each type of test case in the **MT Test Case Types** table, one or more test cases were defined in each **MT Test Cases** table. This included expected results for measurement of the amount of reuse (**Expected Results**) based on the **Software Models Used**. The **Expected Results** were obtained without using the automated version of the measurement framework.
4. Each test case in each **MT Test Cases** table is applied to test measurement of the amount of reuse using the Software Models cited in the row. Actual results obtained using the automated version of the measurement framework to measure the amount of reuse (**Actual Results**) are entered in the same row for the test case.

Variables

Independent Variables

The following independent variables are obtained after execution of the collection process for the measurement testing phase.

Independent variables consist of:

1. The **Software model types** table.
2. The **Implementations** table.
3. The **Software model types and implementations** table.
4. The **Software Model Type Classifications** table.
5. The **MT Test Case Types** table.
6. The **MT Test Case Type ID**, **Test Case Description,** and **Expected Results** in each row of each **MT Test Cases** table.
7. The MLC instances of the Amount of Reuse Measurement M2 Model MLC.
8. The MLC instances of the Software Model Type Classification MLC.
9. The MLC instances of the Software Model Type Set Theory MLC.

10. The MLC instances of the Amount of Reuse Measurement M1 Model Specifier MLC.
11. The Software Model Classification MLC.
12. The Software Model Set Theory MLC.
13. The Amount of Reuse Measurement Model Specifier MLC.

Dependent Variables

The following dependent variables are obtained after execution of the collection process for the measurement testing methodology phase.

Dependent variables consist of:

1. The **Software Models Used** and **Actual Results** in each row of each **MT Test Cases** table.
2. The MLC instances of the Software Model Classification MLC.
3. The MLC instances of the Software Model Set Theory MLC.
4. The MLC instances of the Amount of Reuse Measurement Model Specifier MLC.

Analysis Process

Hypothesis Measures

Base indicators, coverage indicators, and limitation indicators used in this phase are tabulated in Table 4-18, Table 4-19, and Table 4-20. See Appendix C4 for a guide to interpretation of the hypothesis measures used in this methodology phase.

Table 4-18 lists the base indicators used in the measurement testing methodology phase to support calculations for coverage and limitation indicators. Base indicators include the number of software model types, software model type classifications, and test cases for each approach to reuse.

Table 4-18: Base Indicators used in Measurement Testing

Model Type Category	Names of Measures
All Software Model Types	**SMT_BI, SMTI_BI, SMTC_BI, MTTCTR_BI, MTTCTECR_BI, MTTCTICR_BI, MTTCTEGR_BI, MTTCTIGR_BI, MTTC_BI, MTTCEC_BI, MTTCIC_BI, MTTCEG_BI, MTTCIG_BI, MTTCR_BI, MTTCECR_BI, MTTCICR_BI, MTTCEGR_BI, MTTCIGR_BI, MTTCP_BI, MTTCECP_BI, MTTCICP_BI, MTTCEGP_BI, MTTCIGP_BI**
Object-oriented/UML Software Model Types	**OU_SMT_BI, OU_SMTI_BI, OU_SMTC_BI, OU_MTTC_BI, OU_MTTCEC_BI, OU_MTTCIC_BI, OU_MTTCEG_BI, OU_MTTCIG_BI, OU_MTTCR_BI, OU_MTTCECR_BI, OU_MTTCICR_BI, OU_MTTCEGR_BI, OU_MTTCIGR_BI, OU_MTTCP_BI, OU_MTTCECP_BI, OU_MTTCICP_BI, OU_MTTCEGP_BI, OU_MTTCIGP_BI**

Below are six examples of base indicators from the guide to interpretation (Appendix C4).

Name of Measure	Calculation
MTTCTR_BI (MT Test Case Types Base Indicator)	#rows in all **MT Test Case Types** tables
MTTCTECR_BI (MT Test Case Types Base External Composition Indicator)	#rows in **MT Test Case Types** table where **Reuse Approach** = External Composition Reuse
MTTC_BI (MT Test Cases Base Indicator)	#number of **MT Test Cases** tables
MTTCEC_BI (MT Test Cases External Composition Base Indicator)	#number of **MT Test Cases** tables where **Reuse Approach** = External Composition Reuse
MTTCR_BI (MT Test Cases Base Indicator)	#rows in all **MT Test Cases** tables
MTTCECR_BI (MT Test Cases External Composition Base Indicator)	#rows in all **MT Test Cases** tables where **Reuse Approach** = External Composition Reuse
MTTCP_BI (MT Test Cases Pairs Base Indicator)	#pairs of **MT Test Cases** $\text{tables}_{1,2}$ where (**SMTI ID**$_1$ ≠ **SMTI ID**$_2$)
MTTCECP_BI (MT Test Cases Pairs External Composition Base Indicator)	#pairs of **MT Test Cases** $\text{tables}_{1,2}$ where (**Reuse Approach**$_1$ = External Composition Reuse) and (**Reuse Approach**$_2$ = External Composition Reuse) and (**SMTI ID**$_1$ ≠ **SMTI ID**$_2$)

MTTCTR_BI: This indicator is the number of test case types identified for the purposes of this thesis and located as a number of rows in all **MT Test Case Types** table.

MTTCTECR_BI: This indicator is the number of test case types identified for the external composition reuse approach. The value is obtained by counting the number of rows in the **MT Test Case Types** table where the **Reuse Approach** = External Composition.

MTTC_BI: This indicator is the number of software model types with implementations that were used to test measurement of the amount of reuse and is the number of **MT Test Cases** tables.

MTTCEC_BI: This indicator is the number of software model types with implementations that were used to test measurement of the amount of reuse for the external composition reuse approach. The value is obtained by counting the number of **MT Test Cases** tables where the **Reuse Approach** = External Composition Reuse.

MTTCR_BI: This indicator is the total number of test cases for all software model types with implementations that were used to test measurement of the amount of reuse. The number is the total number of rows in all **MT Test Cases** tables.

MTTCECR_BI: This indicator is the total number of test cases for all software model types with implementations that were used to test measurement of the amount of reuse for the external composition reuse approach. The value is obtained by counting the number of rows in **MT Test Cases** tables where the **Reuse Approach** = External Composition Reuse.

MTTCP_BI: This indicator is the number of pairs for all software model types with implementations that were used in measurement testing. The value is obtained by counting all the possible pairs of **MT Test Cases** tables with different values for **SMTI ID** for each table in a pair ($^{\#\text{MT Test Cases tables}}C_2$, or the sum of all values between zero and MTTC_BI subtract MTTC_BI).

MTTCECP_BI: This indicator is the number of pairs for all software model types with implementations that were used in external composition measurement testing. The value is obtained by counting all the possible pairs of **MT Test Cases** tables with the **Reuse Approach** equal to External Composition and different values for **SMTI ID** for each table in a pair ($^{\#\text{MT Test Cases tables = External Composition}}\mathbf{C}_2$, or the sum of all values between zero and MTTCEC_BI subtract MTTCEC_BI).

Table 4-19 lists the coverage indicators used in the measurement testing methodology phase.

Table 4-19: Coverage Indicators used in Measurement Testing

Model Type Category	Names of Measures
All Software Model Types	**Category 1: RMCO_CI, ECRMCO_CI, ICRMCO_CI, EGRMCO_CI, IGRMCO_CI, PSRM_CI, ECPSRM_CI, ICPSRM_CI, EGPSRM_CI, IGPSRM_CI, SRM_CI, ECSRM_CI, ICSRM_CI, EGSRM_CI, IGSRM_CI** **Category 2: URM_CI, ECURM_CI, ICURM_CI, EGURM_CI, IGURM_CI, PURM_CI, ECPURM_CI, ICPURM_CI, EGPURM_CI, IGPURM_CI**
Object-oriented/UML Software Model Types	**Category 1: OU_RMCO_CI, OU_ECRMCO_CI, OU_ICRMCO_CI, OU_EGRMCO_CI, OU_IGRMCO_CI, OU_SRM_CI, OU_ECSRM_CI, OU_ICSRM_CI, OU_EGSRM_CI, OU_IGSRM_CI, OU_URM_CI, OU_PSRM_CI, OU_ECPSRM_CI, OU_ICPSRM_CI, OU_EGPSRM_CI, OU_IGPSRM_CI** **Category 2: OU_ECURM_CI, OU_ICURM_CI, OU_EGURM_CI, OU_IGURM_CI, OU_PURM_CI, OU_ECPURM_CI, OU_ICPURM_CI, OU_EGPURM_CI, OU_IGPURM_CI**

Category 1 indicators measure the level of successful measurement of the amount of reuse for different software models and different approaches to reuse.

Category 2 indicators measure the degree of failure at measurement of the amount of reuse for different software models and different approaches to reuse.

Below are four examples of coverage from the guide to interpretation (Appendix C4). The first and second examples are category 1 indicators that quantify outcome 1 from the experiment overview. The third and fourth examples are category 2 indicators that quantify outcome 2 in the experiment overview.

Name of Measure	Calculation
SRM_CI (Successful Reuse Measurement Coverage Indicator)	#rows in **MT Test Cases** tables where **Expected Results** = **Actual Results**
ECSRM_CI (External Composition Successful Reuse Measurement Coverage Indicator)	#rows in **MT Test Cases** tables where (**Expected Results** = **Actual Results**) and (**Reuse Approach** = External Composition Reuse)
URM_CI (Unsuccessful Reuse Measurement Coverage Indicator)	#rows in **MT Test Cases** tables where **Expected Results** ≠ **Actual Results**
ECURM_CI (External Composition Unsuccessful Reuse Measurement Coverage Indicator)	#rows in **MT Test Cases** tables where (**Expected Results** ≠ **Actual Results**) and (**Reuse Approach** = External Composition Reuse)

SRM_CI: This indicator attempts to express the degree of successful reuse measurement relative to the number of measurement attempts. The value is obtained by counting the number successful reuse measurement attempts of software models for all software model types. This value is compared with MTTCR_BI. A high value indicates that the degree of successful reuse measurement is high. The highest possible value for SRM_CI is MTTCR_BI. A low value indicates that the degree of successful reuse measurement is low. The lowest possible value for SRM_CI is zero.

ECSRM_CI: This indicator attempts to express the degree of successful reuse measurement relative to the number of measurement attempts for the external composition reuse approach. The value is obtained by counting the number successful external composition reuse measurement attempts of software models for all software model types. This value is compared with MTTCECR_BI. A high value indicates that the degree of successful external composition reuse measurement is high. The highest possible value for ECSRM_CI is MTTCECR_BI. A low value indicates that the degree of successful external composition reuse measurement is low. The lowest possible value for ECSRM_CI is zero.

URM_CI: This indicator attempts to express the degree of unsuccessful reuse measurement relative to the number of measurement attempts. The value is obtained by counting the number unsuccessful reuse measurement attempts of software models for all software model types. This value is compared with MTTCR_BI. A high value indicates that the degree of unsuccessful reuse measurement is high. The highest possible value for URM_CI is MTTCR_BI. A low value indicates that the degree of unsuccessful reuse measurement is low. The lowest possible value for URM_CI is zero.

ECURM_CI: This indicator attempts to express the degree of unsuccessful reuse measurement relative to the number of measurement attempts for the external composition reuse approach. The value is obtained by counting the number unsuccessful external composition reuse measurement attempts of software models for all software model types. This value is compared with MTTCECR_BI. A high value indicates that the degree of unsuccessful external composition reuse measurement is high. The highest possible value for ECURM_CI is MTTCECR_BI. A low value indicates that the degree of unsuccessful external composition reuse measurement is low. The lowest possible value for ECURM_CI is zero.

Table 4-20 lists the limitation indicators used in the measurement testing methodology phase.

Table 4-20: Limitation Indicators used in Measurement Testing

Model Type Category	Names of Measures
All Software Model Types	**Category 1: PCSMTI_LI, CSMTI_LI** **Category 2: SCSRM_LI, ECSCSRM_LI, ICSCSRM_LI, EGSCSRM_LI, IGSCSRM_LI, UCSRM_LI, ECUCSRM_LI, ICUCSRM_LI, EGUCSRM_LI, IGUCSRM_LI, UCURM_LI, ECUCURM_LI, ICUCURM_LI, EGUCURM_LI, IGUCURM_LI, SCURM_LI, ECSCURM_LI, ICSCURM_LI, EGSCURM_LI, IGSCURM_LI** **Category 3: SRTCCTC_LI, ECSRTCCTC_LI, ICSRTCCTC_LI, EGSRTCCTC_LI, IGSRTCCTC_LI, SRTCDTC_LI, EGSRTCDTC_LI, IGSRTCDTC_LI, ECSRTCDTC_LI, ICSRTCDTC_LI**
Object-oriented/UML Software Model Types	**Category 1: OU_PCSMTI_LI, OU_CSMTI_LI** **Category 2: OU_SCSRM_LI, OU_ECSCSRM_LI, OU_ICSCSRM_LI, OU_EGSCSRM_LI, OU_IGSCSRM_LI, OU_UCSRM_LI, OU_ECUCSRM_LI, OU_ICUCSRM_LI, OU_EGUCSRM_LI, OU_IGUCSRM_LI, OU_UCURM_LI, OU_ECUCURM_LI, OU_ICUCURM_LI, OU_EGUCURM_LI, OU_IGUCURM_LI, OU_SCURM_LI, OU_ECSCURM_LI, OU_ICSCURM_LI, OU_EGSCURM_LI, OU_IGSCURM_LI** **Category 3: OU_SRTCCTC_LI, OU_ECSRTCCTC_LI, OU_ICSRTCCTC_LI, OU_EGSRTCCTC_LI, OU_IGSRTCCTC_LI, OU_SRTCDTC_LI, OU_ECSRTCDTC_LI, OU_ICSRTCDTC_LI, OU_EGSRTCDTC_LI, OU_IGSRTCDTC_LI**

Category 1 indicators measure the coverage of software model types available relative to the number of attempts at classification of software model types.

Category 2 indicators measure the degree of successful measurement of the amount of reuse relative to the approach to reuse and results for classification of software model types.

Category 3 indicators measure the degree of successful measurement of the amount of reuse relative to the approach to reuse and the similarity between software model type classifications for different software model types.

Below are seven examples of limitation indicators taken from the guide to interpretation (Appendix C4). The first example is a category 1 indicator that quantifies outcome 3 from the experiment overview. The second and third examples are category 2 indicators that quantify outcome 4 from the experiment overview. The

fourth and fifth examples are category 3 indicators that quantify outcome 5 from the experiment overview. The sixth and seventh examples are category 3 indicators that quantify outcome 6 from the experiment overview.

Name of Measure	Calculation
CSMTI_LI (Coverage of Software Model Types with Implementations Limitation Indicator)	#rows in the **Software model type classifications** table where **SMTI ID** = **SMTI ID** in one row in the **Software model types and Implementations** table
SCSRM_LI (Successful Classification Successful Reuse Measurement Limitation Indicator)	#number of **MT Test Cases** tables where (**Actual Classification Scheme** ≠ "NOT ABLE TO CLASSIFY SOFTWARE MODEL TYPE") × (#rows in **MT Test Cases** table where **Expected Results** = **Actual Results**)
ECSCSRM_LI (External Composition Successful Classification Successful Reuse Measurement Limitation Indicator)	#number of **MT Test Cases** tables where ((**Actual Classification Scheme** ≠ "NOT ABLE TO CLASSIFY SOFTWARE MODEL TYPE") and (**Reuse Approach** = External Composition Reuse)) × (#rows in **MT Test Cases** table where **Expected Results** = **Actual Results**)
SRTCCTC_LI (Successful Reuse Test Case Common Type Classification Limitation Indicator)	#pairs of **MT Test Cases** $tables_{1,2}$ where (**SMTI ID**$_1$ ≠ **SMTI ID**$_2$) and (**Actual Classification Scheme**$_1$ = **Actual Classification Scheme**$_2$) and (**Actual Classification Scheme**$_1$ ≠ "NOT ABLE TO CLASSIFY SOFTWARE MODEL TYPE") and (**Actual Classification Scheme**$_2$ ≠ "NOT ABLE TO CLASSIFY SOFTWARE MODEL TYPE") and (#rows in **MT Test Cases** table$_1$ where **Expected Results** = **Actual Results** ≥ 1) and (#rows in **MT Test Cases** table$_2$ where **Expected Results** = **Actual Results** ≥ 1)
ECSRTCCTC_LI (External Composition Successful Reuse Test Case Common Type Classification Limitation Indicator)	#pairs of **MT Test Cases** $tables_{1,2}$ where (**Reuse Approach**$_1$ = External Composition Reuse) and (**Reuse Approach**$_2$ = External Composition Reuse) and (**SMTI ID**$_1$ ≠ **SMTI ID**$_2$) and (**Actual Classification Scheme**$_1$ = **Actual Classification Scheme**$_2$) and (**Actual Classification Scheme**$_1$ ≠ "NOT ABLE TO CLASSIFY SOFTWARE MODEL TYPE") and (**Actual Classification Scheme**$_2$ ≠ "NOT ABLE TO CLASSIFY SOFTWARE MODEL TYPE") and (#rows in **MT Test Cases** table$_1$ where **Expected Results** = **Actual Results** ≥ 1) and (#rows in **MT Test Cases** table$_2$ where **Expected Results** = **Actual Results** ≥ 1)

Name of Measure	Calculation
SRTCDTC_LI (Successful Reuse Test Case Different Type Classification Limitation Indicator)	#pairs of **MT Test Cases** $tables_{1,2}$ where (**SMTI ID**$_1$ ≠ **SMTI ID**$_2$) and (**Actual Classification Scheme**$_1$ ≠ **Actual Classification Scheme**$_2$) and (**Actual Classification Scheme**$_1$ ≠ "NOT ABLE TO CLASSIFY SOFTWARE MODEL TYPE") and (**Actual Classification Scheme**$_2$ ≠ "NOT ABLE TO CLASSIFY SOFTWARE MODEL TYPE") and (#rows in **MT Test Cases** $table_1$ where **Expected Results** = **Actual Results** ≥ 1) and (#rows in **MT Test Cases** $table_2$ where **Expected Results** = **Actual Results** ≥ 1)
ECSRTCDTC_LI (External Composition Successful Reuse Test Case Different Type Classification Limitation Indicator)	#pairs of **MT Test Cases** $tables_{1,2}$ where (**Reuse Approach**$_1$ = External Composition Reuse) and (**Reuse Approach**$_2$ = External Composition Reuse) and (**SMTI ID**$_1$ ≠ **SMTI ID**$_2$) and (**Actual Classification Scheme**$_1$ ≠ **Actual Classification Scheme**$_2$) and (**Actual Classification Scheme**$_1$ ≠ "NOT ABLE TO CLASSIFY SOFTWARE MODEL TYPE") and (**Actual Classification Scheme**$_2$ ≠ "NOT ABLE TO CLASSIFY SOFTWARE MODEL TYPE") and (#rows in **MT Test Cases** $table_1$ where **Expected Results** = **Actual Results** ≥ 1) and (#rows in **MT Test Cases** $table_2$ where **Expected Results** = **Actual Results** ≥ 1)

CSMTI_LI: This indicator attempts to assess the number of classification attempts relative to the number of software model types with implementations. The value is calculated by counting the number of software model types and implementations where a classification attempt is made. The value is compared with SMTI_BI. A high value indicates many classification attempts made relative to software model types with implementations. The highest possible value for CSMTI_LI is equal to SMTI_BI. A low value indicates few classification attempts made relative to software model types with implementations. The lowest possible value for CSMTI_LI is equal to zero.

SCSRM_LI: This indicator attempts to assess the degree of successful reuse measurement for software model types with successful software model type classifications. The value is obtained by counting the number of successful software model reuse measurements for all Software model types that have a successful software model type classification. This value is compared to MTTCR_BI. A high value indicates a high degree of successful software model measurement for software

model types with successful software model type classifications. The highest possible value for SCSRM_LI is MTTCR_BI. A low value indicates a low degree of successful software model measurement for software model types with successful software model type classifications. The lowest possible value for SCSRM_LI is zero.

ECSCSRM_LI: This indicator attempts to assess the degree of successful reuse measurement for software model types with successful software model type classifications for the external composition reuse approach. The value is obtained by counting the number of successful external composition reuse measurements for all Software model types that have a successful software model type classification. This value is compared to MTTCECR_BI. A high value indicates a high degree of successful external composition reuse measurement for software model types with successful software model type classifications. The highest possible value for ECSCSRM_LI is MTTCECR_BI. A low value indicates a low degree of successful external composition reuse measurement for software model types with successful software model type classifications. The lowest possible value for ECSCSRM_LI is zero.

SRTCCTC_LI: This indicator attempts to assess the degree of common software model type classification for different software model types with successful software model type classifications and at least one successful attempt at reuse measurement. The value is obtained by counting the total number of software model type pairs that are different in name but have the same successful software model type classification and at least one successful attempt at reuse measurement. The value is compared to MTTCP_BI. A high value indicates a high degree of common software model type classification. The highest possible value for SRTCCTC_LI is equal to MTTCP_BI. A low value indicates a low degree of common software model type classification. The lowest possible value for SRTCCTC_LI is zero.

ECSRTCCTC_LI: This indicator attempts to assess the degree of common software model type classification for different software model types with successful software model type classifications and at least one successful attempt at external composition reuse measurement. The value is obtained by counting the total number of software model type pairs that are different in name but have the same successful software model type classification and at least one successful attempt at external composition reuse measurement. The value is compared to MTTCECP_BI. A high value indicates a high degree of common software model type classification for the external composition reuse approach. The highest possible value for ECSRTCCTC_LI is equal to MTTCECP_BI. A low value indicates a low degree of common software

model type classification for the external composition reuse approach. The lowest possible value for ECSRTCCTC_LI is zero.

SRTCDTC_LI: This indicator attempts to assess the degree of different software model type classification for different software model types with successful software model type classifications and at least one successful attempt at reuse measurement. The value is obtained by counting the total number of software model type pairs that are different in name and have a different successful software model type classification and at least one successful attempt at reuse measurement. The value is compared to MTTCP_BI. A high value indicates a high degree of different software model type classification. The highest possible value for SRTCDTC_LI is equal to MTTCP_BI. A low value indicates a low degree of different software model type classification. The lowest possible value for SRTCDTC_LI is zero.

ECSRTCDTC_LI: This indicator attempts to assess the degree of different software model type classification for different software model types with successful software model type classifications and at least one successful attempt at external composition reuse measurement. The value is obtained by counting the total number of software model type pairs that are different in name and have a different successful software model type classification and at least one successful attempt at external composition reuse measurement. The value is compared to MTTCECP_BI. A high value indicates a high degree of different software model type classification for the external composition reuse approach. The highest possible value for ECSRTCDTC_LI is equal to MTTCECP_BI. A low value indicates a low degree of different software model type classification for the external composition reuse approach. The lowest possible value for ECSRTCDTC_LI is zero.

Assessment Criteria

Hypothesis measures used to test the hypotheses is illustrated in:

- Table 4-21 for **H0a** and **H0b**.
- Table 4-22 for **H1** via **SH 1.5**, **SH 1.6**, **SH 1.7**, **SH 1.8**, **SH 1.9**, **SH 1.10**, **SH 1.11**, and **SH 1.12**.
- Table 4-23 for **H2** via **SH 2.5**, **SH 2.6**, **SH 2.7**, **SH 2.8**, **SH 2.9**, **SH 2.10**, **SH 2.11**, and **SH 2.12**.

Refer to Appendix C4 for further details on descriptions for assessment criteria.

Table 4-21 names the hypothesis measures used for testing the null hypothesis **H0a** and **H0b**.

Table 4-21: Assessment of **H0a/b** using hypothesis measures

Hypothesis Part	Measures Used
H0a	**PSRM_CI, ICPSRM_CI, ECPSRM_CI, IGPSRM_CI, EGPSRM_CI, PURM_CI, ICPURM_CI, ECPURM_CI, IGPURM_CI, EGPURM_CI OU_PSRM_CI, OU_ICPSRM_CI, OU_ECPSRM_CI, OU_IGPSRM_CI, OU_EGPSRM_CI, OU_PURM_CI, OU_ICPURM_CI, OU_ECPURM_CI, OU_IGPURM_CI, OU_EGPURM_CI**
H0b	**PCSMTI_LI, CSMTI_LI SCSRM_LI, UCSRM_LI, UCURM_LI, SCURM_LI SRTCCTC_LI, SRTCDTC_LI OU_PCSMTI_LI, OU_CSMTI_LI OU_SCSRM_LI, OU_UCSRM_LI, OU_MTTCR_LI, OU_UCURM_LI, OU_SCURM_LI OU_SRTCCTC_LI, OU_SRTCDTC_LI**

How the hypothesis measures support **H0a/b** is detailed in the guide to interpretation (Appendix C4). Below are some examples from the guide to interpretation that illustrate this.

Hypothesis Part	Name of Measure	Value to Support Hypothesis	Value to Deny Hypothesis
H0a	**PSRM_CI**	PSRM_CI = 0	PSRM_CI > 0
H0b	**CSMTI_LI, SMTI_BI**	CSMTI_LI = SMTI_BI and SMTI_BI >= 1	CSMTI_LI < SMTI_BI and SMTI_BI >= 1
	SCSRM_LI, MTTCR_BI	SCSRM_LI = MTTCR_BI	SCSRM_LI < MTTCR_BI

The truth of **H0a** is *supported* if:

- There are no successful attempts at measuring reuse (PSRM_CI = 0).

To *deny* **H0a**:

- There must be some successful attempts at measuring reuse (PSRM_CI > 0).

The truth of **H0b** is *supported* if:

- All software model types with an implementation must have at least one attempt at classification of the software model type (CSMTI_LI = SMTI_BI) and there must be at least one software model type with an implementation included in the study (SMTI_BI >= 1).
- All test cases must have successful attempts at measurement of reuse for all software model types with successful software model type classifications (SCSRM_LI = MTTCR_BI).

To *deny* **H0b**:

- Some software model types with an implementation do not have one single attempt at classification of the software model type (CSMTI_LI < SMTI_BI) and there must be at least one software model type with an implementation included in the study (SMTI_BI >= 1).
- Not all test cases have successful attempts at measurement of reuse for all software model types with successful software model type classifications (SCSRM_LI < MTTCR_BI).

Table 4-22 names the hypothesis measures used for testing hypothesis **H1** via **SH 1.5**, **SH 1.6**, **SH 1.7**, **SH 1.8**, **SH 1.9**, **SH 1.10**, **SH 1.11**, and **SH 1.12**.

Table 4-22: Assessment of Hypothesis **H1** using Hypothesis measures and **SH 1.5**, **SH 1.6**, **SH 1.7**, **SH 1.8**, **SH 1.9**, **SH 1.10**, **SH 1.11**, and **SH 1.12**.

Sub-Hypothesis	Measures Used
SH 1.5	RMCO_CI, SRM_CI, URM_CI, PSRM_CI, PURM_CI, ICRMCO_CI, ICSRM_CI, ICURM_CI, ICPSRM_CI, ICPURM_CI, MTTC_BI, MTTCIC_BI OU_RMCO_CI, OU_SRM_CI, OU_URM_CI, OU_PSRM_CI, OU_PURM_CI, OU_ICRMCO_CI, OU_ICSRM_CI, OU_ICURM_CI, OU_ICPSRM_CI, OU_ICPURM_CI, OU_MTTC_BI, OU_MTTCIC_BI
SH 1.6	RMCO_CI, SRM_CI, URM_CI, PSRM_CI, PURM_CI, ECRMCO_CI, ECSRM_CI, ECURM_CI, ECPSRM_CI, ECPURM_CI, MTTC_BI, MTTCEC_BI OU_RMCO_CI, OU_SRM_CI, OU_URM_CI, OU_PSRM_CI, OU_PURM_CI, OU_ECRMCO_CI, OU_ECSRM_CI, OU_ECURM_CI, OU_ECPSRM_CI, OU_ECPURM_CI, OU_MTTC_BI, OU_MTTCEC_BI
SH 1.7	RMCO_CI, SRM_CI, URM_CI, PSRM_CI, PURM_CI, IGRMCO_CI, IGSRM_CI, MTTCIG_BI, IGURM_CI, IGPSRM_CI, IGPURM_CI, MTTC_BI, MTTCIG_BI OU_RMCO_CI, OU_SRM_CI, OU_URM_CI, OU_PSRM_CI, OU_PURM_CI, OU_IGRMCO_CI, OU_IGSRM_CI, OU_MTTCIG_BI, OU_IGURM_CI, OU_IGPSRM_CI, OU_IGPURM_CI, OU_MTTC_BI, OU_MTTCIG_BI
SH 1.8	RMCO_CI, SRM_CI, URM_CI, PSRM_CI, PURM_CI, EGRMCO_CI, EGSRM_CI, EGURM_CI, EGPSRM_CI, EGPURM_CI, MTTC_BI, MTTCEG_BI OU_RMCO_CI, OU_SRM_CI, OU_URM_CI, OU_PSRM_CI, OU_PURM_CI, OU_EGRMCO_CI, OU_EGSRM_CI, OU_EGURM_CI, OU_EGPSRM_CI, OU_EGPURM_CI, OU_MTTC_BI, OU_MTTCEG_BI
SH 1.9	RMCO_CI, SRM_CI, URM_CI, PSRM_CI, PURM_CI, ICRMCO_CI, ICSRM_CI, ICURM_CI, ICPSRM_CI, ICPURM_CI, MTTCR_BI, MTTCICR_BI OU_RMCO_CI, OU_SRM_CI, OU_URM_CI, OU_PSRM_CI, OU_PURM_CI, OU_ICRMCO_CI, OU_ICSRM_CI, OU_ICURM_CI, OU_ICPSRM_CI, OU_ICPURM_CI, OU_MTTCR_BI, OU_MTTCICR_BI
SH 1.10	RMCO_CI, SRM_CI, URM_CI, PSRM_CI, PURM_CI, ECRMCO_CI, ECSRM_CI, ECURM_CI, ECPSRM_CI, ECPURM_CI, MTTCR_BI, MTTCECR_BI OU_RMCO_CI, OU_SRM_CI, OU_URM_CI, OU_PSRM_CI, OU_PURM_CI, OU_ECRMCO_CI, OU_ECSRM_CI, OU_ECURM_CI, OU_ECPSRM_CI, OU_ECPURM_CI, OU_MTTCR_BI, OU_MTTCECR_BI
SH 1.11	RMCO_CI, SRM_CI, URM_CI, PSRM_CI, PURM_CI, IGRMCO_CI, IGSRM_CI, IGURM_CI, IGPSRM_CI, IGPURM_CI, MTTCR_BI, MTTCIGR_BI OU_RMCO_CI, OU_SRM_CI, OU_URM_CI, OU_PSRM_CI, OU_PURM_CI, OU_IGRMCO_CI, OU_IGSRM_CI, OU_IGURM_CI, OU_IGPSRM_CI, OU_IGPURM_CI, OU_MTTCR_BI, OU_MTTCIGR_BI
SH 1.12	RMCO_CI, SRM_CI, URM_CI, PSRM_CI, PURM_CI, EGRMCO_CI, EGSRM_CI, EGURM_CI, EGPSRM_CI, EGPURM_CI, MTTCR_BI, MTTCEGR_BI OU_RMCO_CI, OU_SRM_CI, OU_URM_CI, OU_PSRM_CI, OU_PURM_CI, OU_EGRMCO_CI, OU_EGSRM_CI, OU_EGURM_CI, OU_EGPSRM_CI, OU_EGPURM_CI, OU_MTTCR_BI, OU_MTTCEGR_BI

How the hypothesis measures support H1 is detailed in the guide to interpretation (Appendix C4). Below are some examples from the guide to interpretation that illustrate this.

Sub-Hypothesis	Name of Measure	Value to Support Hypothesis	Value to Not Support Hypothesis
SH 1.6	**SRM_CI MTTC_BI**	SRM_CI > 1 and MTTC_BI > 1	SRM_CI < 2 and MTTC_BI > 1
	URM_CI, MTTC_BI	URM_CI + 1 < MTTCR_BI and MTTC_BI > 1	URM_CI + 1 >= MTTCR_BI and MTTC_BI > 1
	ECSRM_CI MTTCEC_BI	ECSRM_CI > 1 and MTTCEC_BI > 1	ECSRM_CI < 2 and MTTCEC_BI > 1
	ECURM_CI, MTTCEC_BI	ECURM_CI + 1 < MTTCECR_BI and MTTCEC_BI > 1	ECURM_CI + 1 >= MTTCECR_BI and MTTCEC_BI > 1
SH 1.10	**SRM_CI MTTCR_BI**	SRM_CI > 1 and MTTCR_BI > 1	SRM_CI < 2 and MTTCR_BI > 1
	URM_CI, MTTCR_BI	URM_CI + 1 < MTTCR_BI and MTTCR_BI > 1	URM_CI + 1 >= MTTCR_BI and MTTCR_BI > 1
	ECSRM_CI MTTCECR_BI	ECSRM_CI > 1 and MTTCECR_BI > 1	ECSRM_CI < 2 and MTTCECR_BI > 1
	ECURM_CI, MTTCECR_BI	ECURM_CI + 1 < MTTCECR_BI and MTTCECR_BI > 1	ECURM_CI + 1 >= MTTCECR_BI and MTTCECR_BI > 1

The truth of **H1** is *supported* if the truth of **SH 1.6** and **SH 1.10** are *supported.*

To *support* **SH 1.6**:

- There must be at least two successful attempts at measurement of reuse (SRM_CI > 1) with at least two software model types tested (MTTC_BI > 1).
- The number of unsuccessful attempts at measurement of reuse must be at least two less than the number of attempts at measurement of reuse (URM_CI + 1 < MTTCR_BI) with at least two software model types tested (MTTC_BI > 1).
- There must be at least two successful attempts at measurement of internal composition reuse (ECSRM_CI > 1) with at least two software model types tested (MTTCEC_BI > 1).
- The number of unsuccessful attempts at measurement of internal composition reuse must be at least two less than the number of attempts at measurement of internal composition reuse (ECURM_CI + 1 < MTTCECR_BI) with at least two software model types tested (MTTCEC_BI > 1).

SH 1.6 is *not supported* if:

- There is only one successful attempt at measurement of reuse (SRM_CI < 2) with at least two software model types tested (MTTC_BI > 1).
- The number of unsuccessful attempts at measurement of reuse plus one is equal to or more than the number of attempts at measurement of reuse (URM_CI + 1 >= MTTCR_CI) with at least two software model types tested (MTTC_BI > 1).
- There is only one successful attempt at measurement of external composition reuse (ECSRM_CI < 2) with at least two software model types tested (MTTCEC_BI > 1).
- The number of unsuccessful attempts at measurement of external composition reuse plus one is equal to or more than the number of attempts at measurement of external composition reuse (ECURM_CI + 1 >= MTTCECR_CI) with at least two software model types tested (MTTCEC_BI > 1).

To *support* **SH 1.10**:

- There must be at least two successful attempts at measurement of reuse (SRM_CI > 1) with at least two attempts at measurement of reuse (MTTCR_BI > 1).
- The number of unsuccessful attempts at measurement of reuse must be at least two less than the number of attempts at measurement of reuse (URM_CI + 1 < MTTCR_BI) with at least two attempts at measurement of reuse (MTTCR_BI > 1).
- There must be at least two successful attempts at measurement of external composition reuse (ECSRM_CI > 1) with at least two attempts at measurement of external composition reuse (MTTCECR_BI > 1).
- The number of unsuccessful attempts at measurement of external composition reuse must be at least two less than the number of attempts at measurement of external composition reuse (ECURM_CI + 1 < MTTCECR_BI) with at least two attempts at measurement of external composition reuse (MTTCECR_BI > 1).

SH 1.10 is *not supported* if:

- There is only one successful attempt at measurement of reuse (SRM_CI < 2) with at least two attempts at measurement of reuse (MTTCR_BI > 1).
- The number of unsuccessful attempts at measurement of reuse plus one is equal to or more than the number of attempts at measurement of reuse (URM_CI + 1 >= MTTCR_CI) with at least two attempts at measurement of reuse (MTTCR_BI > 1).
- There is only one successful attempt at measurement of external composition reuse (ECSRM_CI < 2) with at least two attempts at measurement of external composition reuse (MTTCECR_BI > 1).
- The number of unsuccessful attempts at measurement of external composition reuse plus one is equal to or more than the number of attempts at measurement of external composition reuse (ECURM_CI + 1 >= MTTCECR_CI) with at least two attempts at measurement of external composition reuse (MTTCECR_BI > 1).

Table 4-23 names the hypothesis measures used for testing hypothesis **H2** via **SH 2.5**, **SH 2.6**, **SH 2.7**, **SH 2.8**, **SH 2.9**, **SH 2.10**, **SH 2.11**, and **SH 2.12**.

Table 4-23: Assessment of Hypothesis **H2** using Hypothesis measures and **SH 2.5, SH 2.6, SH 2.7, SH 2.8, SH 2.9, SH 2.10, SH 2.11,** and **SH 2.12.**

Sub-Hypothesis	Measures Used
SH 2.5	PCSMT_LI, CSMTI_LI, SCSRM_LI, UCSRM_LI, UCURM_LI, SCURM_LI, SRTCCTC_LI, SRTCDTC_LI, ICSCRM_LI, ICUCSRM_LI, ICUCURM_LI, ICSCURM_LI, ICSRTCCTC_LI, ICSRTCDTC_LI, MTTC_BI, MTTCIC_BI OU_PCSMT_LI, OU_CSMTI_LI, OU_SCSRM_LI, OU_UCSRM_LI, OU_UCURM_LI, OU_SCURM_LI, OU_SRTCCTC_LI, OU_SRTCDTC_LI, OU_ICSCRM_LI, OU_ICUCSRM_LI, OU_ICUCURM_LI, OU_ICSCURM_LI, OU_ICSRTCCTC_LI, OU_ICSRTCDTC_LI, OU_MTTC_BI, OU_MTTCIC_BI
SH 2.6	PCSMT_LI, CSMTI_LI, SCSRM_LI, UCSRM_LI, UCURM_LI, SCURM_LI, SRTCCTC_LI, SRTCDTC_LI, ECSCRM_LI, ECUCSRM_LI, ECUCURM_LI, ECSCURM_LI, ECSRTCCTC_LI, ECSRTCDTC_LI, MTTC_BI, MTTCEC_BI OU_PCSMT_LI, OU_CSMTI_LI, OU_SCSRM_LI, OU_UCSRM_LI, OU_UCURM_LI, OU_SCURM_LI, OU_SRTCCTC_LI, OU_SRTCDTC_LI, OU_ECSCRM_LI, OU_ECUCSRM_LI, OU_ECUCURM_LI, OU_ECSCURM_LI, OU_ECSRTCCTC_LI, OU_ECSRTCDTC_LI, OU_MTTC_BI, OU_MTTCEC_BI
SH 2.7	PCSMT_LI, CSMTI_LI, SCSRM_LI, UCSRM_LI, UCURM_LI, SCURM_LI, SRTCCTC_LI, SRTCDTC_LI, IGSCRM_LI, IGUCSRM_LI, IGUCURM_LI, IGSCURM_LI, IGSRTCCTC_LI, IGSRTCDTC_LI, MTTC_BI, MTTCIG_BI OU_PCSMT_LI, OU_CSMTI_LI, OU_SCSRM_LI, OU_UCSRM_LI, OU_UCURM_LI, OU_SCURM_LI, OU_SRTCCTC_LI, OU_SRTCDTC_LI, OU_IGSCRM_LI, OU_IGUCSRM_LI, OU_IGUCURM_LI, OU_IGSCURM_LI, OU_IGSRTCCTC_LI, OU_IGSRTCDTC_LI, OU_MTTC_BI, OU_MTTCIG_BI
SH 2.8	SCSRM_LI, UCSRM_LI, UCURM_LI, SCURM_LI, SRTCCTC_LI, SRTCDTC_LI, EGSCRM_LI, EGUCSRM_LI, EGUCURM_LI, EGSCURM_LI, EGSRTCCTC_LI, EGSRTCDTC_LI, MTTC_BI, MTTCEG_BI OU_SCSRM_LI, OU_UCSRM_LI, OU_UCURM_LI, OU_SCURM_LI, OU_SRTCCTC_LI, OU_SRTCDTC_LI, OU_EGSCRM_LI, OU_EGUCSRM_LI, OU_EGUCURM_LI, OU_EGSCURM_LI, OU_EGSRTCCTC_LI, OU_EGSRTCDTC_LI, OU_MTTC_BI, OU_MTTCEG_BI
SH 2.9	SCSRM_LI, UCSRM_LI, UCURM_LI, SCURM_LI, SRTCCTC_LI, SRTCDTC_LI, ICSCRM_LI, ICUCSRM_LI, ICUCURM_LI, ICSCURM_LI, MTTCR_BI, MTTCICR_BI OU_SCSRM_LI, OU_UCSRM_LI, OU_UCURM_LI, OU_SCURM_LI, OU_SRTCCTC_LI, OU_SRTCDTC_LI, OU_ICSCRM_LI, OU_ICUCSRM_LI, OU_ICUCURM_LI, OU_ICSCURM_LI, OU_MTTCR_BI, OU_MTTCICR_BI
SH 2.10	SCSRM_LI, UCSRM_LI, UCURM_LI, SCURM_LI, SRTCCTC_LI, SRTCDTC_LI, ECSCRM_LI, ECUCSRM_LI, ECUCURM_LI, ECSCURM_LI, MTTCR_BI, MTTCECR_BI OU_SCSRM_LI, OU_UCSRM_LI, OU_UCURM_LI, OU_SCURM_LI, OU_SRTCCTC_LI, OU_SRTCDTC_LI, OU_ECSCRM_LI, OU_ECUCSRM_LI, OU_ECUCURM_LI, OU_ECSCURM_LI, OU_MTTCR_BI, OU_MTTCECR_BI
SH 2.11	SCSRM_LI, UCSRM_LI, UCURM_LI, SCURM_LI, SRTCCTC_LI, SRTCDTC_LI, IGSCRM_LI, IGUCSRM_LI, IGUCURM_LI, IGSCURM_LI, MTTCR_BI, MTTCIGR_BI OU_SCSRM_LI, OU_UCSRM_LI, OU_UCURM_LI, OU_SCURM_LI, OU_SRTCCTC_LI, OU_SRTCDTC_LI, OU_IGSCRM_LI, OU_IGUCSRM_LI, OU_IGUCURM_LI, OU_IGSCURM_LI, OU_MTTCR_BI, OU_MTTCIGR_BI
SH 2.12	SCSRM_LI, UCSRM_LI, UCURM_LI, SCURM_LI, SRTCCTC_LI, SRTCDTC_LI, EGSCRM_LI, EGUCSRM_LI, EGUCURM_LI, EGSCURM_LI, MTTCR_BI, MTTCEGR_BI OU_SCSRM_LI, OU_UCSRM_LI, OU_UCURM_LI, OU_SCURM_LI, OU_SRTCCTC_LI, OU_SRTCDTC_LI, OU_EGSCRM_LI, OU_EGUCSRM_LI, OU_EGUCURM_LI, OU_EGSCURM_LI, OU_MTTCR_BI, OU_MTTCEGR_BI

How the hypothesis measures support H2 is detailed in the guide to interpretation (Appendix C4). Below are some examples from the guide to interpretation that illustrate this.

Sub-Hypothesis	Name of Measure	Value to Support Hypothesis	Value to Not Support Hypothesis
SH 2.6	**CSMTI_LI, SMTI_BI**	CSMTI_LI < SMTI_BI and SMTI_BI > 0	CSMTI_LI = SMTI_BI and SMTI_BI > 0
	SCSRM_LI, MTTC_BI	SCSRM_LI < MTTCR_BI and MTTC_BI > 1	SCSRM_LI = MTTCR_BI and MTTC_BI > 1
	SRTCCTC_LI, MTTC_BI	SRTCCTC_LI < MTTCP_BI and MTTC_BI > 1	SRTCCTC_LI = MTTCP_BI and MTTC_BI > 1
	SRTCDTC_LI, MTTC_BI	SRTCDTC_LI > 0 and MTTC_BI > 1	SRTCDTC_LI = 0 and MTTC_BI > 1
	ECSCRM_LI, MTTCEC_BI	ECSCSRM_LI < MTTCECR_BI and MTTCEC_BI > 1	ECSCSRM_LI = MTTCECR_BI and MTTCEC_BI > 1
	ECSRTCCTC_LI, MTTCEC_BI	ECSRTCCTC_LI < MTTCECP_BI and MTTCEC_BI > 1	ECSRTCCTC_LI = MTTCECP_BI and MTTCEC_BI > 1
	ECSRTCDTC_LI, MTTCEC_BI	ECSRTCDTC_LI > 0 and MTTCEC_BI > 1	ECSRTCDTC_LI = 0 and MTTCEC_BI > 1
SH 2.10	**SCSRM_LI, MTTCR_BI**	SCSRM_LI < MTTCR_BI and MTTCR_BI > 1	SCSRM_LI = MTTCR_BI and MTTCR_BI > 1
	ECSCSRM_LI, MTTCECR_BI	ECSCSRM_LI < MTTCECR_BI and MTTCECR_BI > 1	ECSCSRM_LI = MTTCECR_BI and MTTCECR_BI > 1

The truth of **H2** is *supported* if the truth of **SH 2.6** and **SH 2.10** are *supported.*

To *support* **SH 2.6**:

- Some software model types with an implementation do not have one single attempt at classification of the software model type (CSMTI_LI < SMTI_BI) and there must be at least one software model type with an implementation included in the study (SMTI_BI > 0).
- Not all test cases have successful attempts at measurement of reuse for all software model types with successful software model type classifications (SCSRM_LI < MTTCR_BI) with at least two software model types included in the study (MTTC_BI > 1).
- Not all software model types with at least one successful attempt at measurement of reuse used a common software model type classification scheme (SRTCCTC_LI < MTTCP_BI) with at least two software model types included in the study (MTTC_BI > 1).

- Some software model types with at least one successful attempt at measurement of reuse used a different software model type classification scheme (SRTCDTC_LI > 0) with at least two software model types included in the study (MTTC_BI > 1).
- Not all test cases have successful attempts at measurement of external composition reuse for all software model types with successful software model type classifications (ECSCSRM_LI < MTTCR_BI) with at least two software model types included in the study (MTTCEC_BI > 1).
- Not all software model types with at least one successful attempt at measurement of external composition reuse used a common software model type classification scheme (ECSRTCCTC_LI < MTTCECP_BI) with at least two software model types included in the study (MTTCEC_BI > 1).
- Some software model types with at least one successful attempt at measurement of external composition reuse used a different software model type classification scheme (ECSRTCDTC_LI > 0) with at least two software model types included in the study (MTTCEC_BI > 1).

SH 2.6 is *not supported* if:

- All software model types with an implementation must have at least one attempt at classification of the software model type (CSMTI_LI = SMTI_BI) and there must be at least one software model type with an implementation included in the study (SMTI_BI > 0).
- All test cases must have successful attempts at measurement of reuse for all software model types with successful software model type classifications (SCSRM_LI = MTTCR_BI) with at least two software model types included in the study (MTTC_BI > 1).
- All software model types with at least one successful attempt at measurement of reuse used a common software model type classification scheme (SRTCCTC_LI = MTTCP_BI) with at least two software model types included in the study (MTTC_BI > 1).
- No software model types with at least one successful attempt at measurement of reuse used a different software model type classification scheme (SRTCDTC_LI = 0) with at least two software model types included in the study (MTTC_BI > 1).
- All test cases must have successful attempts at measurement of external composition reuse for all software model types with successful software model

type classifications (ECSCSRM_LI = MTTCECR_BI) with at least two software model types included in the study (MTTCEC_BI > 1).

- All software model types with at least one successful attempt at measurement of external composition reuse used a common software model type classification scheme (ECSRTCCTC_LI = MTTCECP_BI) with at least two software model types included in the study (MTTCEC_BI > 1).
- No software model types with at least one successful attempt at measurement of external.composition reuse used a different software model type classification scheme (ECSRTCDTC_LI = 0) with at least two software model types included in the study (MTTCEC_BI > 1).

To *support* **SH 2.10**:

- Not all test cases have successful attempts at measurement of reuse for all software model types with successful software model type classifications (SCSRM_LI < MTTCR_BI) with at least two attempts at measurement of reuse in the study (MTTCR_BI > 1).
- Not all test cases have successful attempts at measurement of external composition reuse for all software model types with successful software model type classifications (ECSCSRM_LI < MTTCECR_BI) with at least two attempts at measurement of external composition reuse in the study (MTTCECR_BI > 1).

SH 2.10 is *not supported* if:

- All test cases must have successful attempts at measurement of reuse for all software model types with successful software model type classifications (SCSRM_LI = MTTCR_BI) with at least two attempts at measurement of reuse in the study (MTTCR_BI > 1).
- All test cases must have successful attempts at measurement of external composition reuse for all software model types with successful software model type classifications (ECSCSRM_LI = MTTCECR_BI) with at least two attempts at measurement of external composition reuse in the study (MTTCECR_BI > 1).

To summarise, **H0a/b** is *supported* if:

- all attempts at measurement of the amount of reuse with different software models fail (**H0a**), and
- all software model types available are classified (**H0b**), and
- all attempts at measurement of the amount of reuse with different software models are successful regardless of the results of classification of software model types (**H0b**), and
- all successful attempts at measurement of the amount of reuse with different software models use the same software model type classification (**H0b**), and

H0 is *denied* if:

- at least one attempt at measurement of the amount of reuse with different software models is successful (**H0a**), or
- not all software model types available are classified (**H0b**), or
- at least one attempt at measurement of the amount of reuse with different software models is successful because of the results of classification of software model types (**H0b**), and
- at least two successful attempts at measurement of the amount of reuse with different software models use a different software model type classification (**H0b**)

H1 is *supported* if:

- Some attempts at measurement of the amount of reuse with different software model types are successful (SH 1.5 – SH 1.12).
- Some attempts at measurement of the amount of reuse, for a particular approach to reuse, with different software model types or software models are successful (internal composition SH 1.5, SH 1.9; external composition SH 1.6, SH 1.10; internal generation SH 1.7, SH 1.11; external generation SH 1.8, SH 1.12).

H1 is *not supported* if:

- All attempts at measurement of the amount of reuse with different software model types fail (SH 1.5 – SH 1.12).
- All attempts at measurement of the amount of reuse, for any particular approach to reuse, with different software model types or software models fail (internal composition SH 1.5, SH 1.9; external composition SH 1.6, SH 1.10; internal generation SH 1.7, SH 1.11; external generation SH 1.8, SH 1.12).

H2 is *supported* if:

- Not all software model types are classified (SH 2.5 – SH 2.12)
- Some attempts at measurement of the amount of reuse with different software models are successful because of the results for classification of software model types (SH 2.5 – SH 2.12).
- Some successful attempts at measurement of the amount of reuse with different software model types use different software model type classifications (SH 2.5 – SH 2.12).
- Some attempts at measurement of the amount of reuse, for a particular approach to reuse, with different software model types or software models are successful because of the results for classification of software model types (internal composition SH 2.5, SH 2.9; external composition SH 2.6, SH 2.10; internal generation SH 2.7, SH 2.11; external generation SH 2.8, SH 2.12).
- Some successful attempts at measurement of the amount of reuse, for a particular approach to reuse, with different software model types use different software model type classifications (internal composition SH 2.5, SH 2.9; external composition SH 2.6, SH 2.10; internal generation SH 2.7, SH 2.11; external generation SH 2.8, SH 2.12).

H2 is *not supported* if:

- All software model types are classified (SH 2.5 – SH 2.12)
- All attempts at measurement of the amount of reuse with different software models are successful regardless of the results for classification of software model types (SH 2.5 – SH 2.12).
- All successful attempts at measurement of the amount of reuse with different software model types use the same software model type classification (SH 2.5 – SH 2.12).
- All attempts at measurement of the amount of reuse, for a particular approach to reuse, with different software model types or software models are successful regardless of the results for classification of software model types (internal composition SH 2.5, SH 2.9; external composition SH 2.6, SH 2.10; internal generation SH 2.7, SH 2.11; external generation SH 2.8, SH 2.12).
- All successful attempts at measurement of the amount of reuse, for a particular approach to reuse, with different software model types use the same software model type classifications (internal composition SH 2.5, SH 2.9; external

composition SH 2.6, SH 2.10; internal generation SH 2.7, SH 2.11; external generation SH 2.8, SH 2.12).

This concludes the assessment criteria in the measurement testing methodology phase for hypotheses **H0a** and **H0b**, **H1/SH 1.5 - 1.12**, and **H2/SH 2.5 - 2.12**.

Analysis Report Input

The methodology phase analysis reports used in this phase are:

The Measurement Testing Analysis Report. See the section under the heading "Measurement Testing Methodology Phase Analysis Report" in Appendix D3 for a complete example.

The hypothesis analysis reports used in this phase are:

The H0 Analysis Report. See the section under the heading "H0 Hypothesis Analysis Report" in Appendix D5 for a complete example.

The H1 Analysis Report. See the section under the heading "H1 Hypothesis Analysis Report" in Appendices D6 – D7 for a complete example.

The H2 Analysis Report. See the section under the heading "H2 Hypothesis Analysis Report" in Appendices D8 – D9 for a complete example.

Analysis Report Output

The analysis reports generated by this methodology phase are:

The Measurement Testing Analysis Report. This report is generated by adding all measures for independent and dependent variables in the measurement testing methodology phase along with the values obtained. In addition, values for all hypothesis measures are added to the report.

The analysis reports modified in this methodology phase are:

The H0 Analysis Report.

The H1 Analysis Report.

The H2 Analysis Report.

The above hypothesis analysis reports include the values obtained for all hypothesis measures from the software model classification methodology phase. Identification of the level of support for each hypothesis is also included and relevant totals for sub-hypotheses and the hypothesis are updated.

Analysis Process Description

1. Calculate the values of the measures derived from the independent variables and dependent variables for the measurement testing methodology phase.
2. Enter the names and values for these measures in the Measurement Testing Methodology Phase Analysis Report.
3. Calculate the values of the base indicators, coverage indicators, and limitation indicators for hypothesis measures listed under hypothesis measures for the measurement testing methodology phase.
4. Enter the values of the hypothesis measures in the appropriate rows of the Measurement Testing Methodology Phase Analysis report.
5. Copy the values for the hypothesis measures in the Measurement Testing Methodology Phase Analysis Report into the appropriate rows in the hypothesis analysis reports. For each of these rows enter the MP_ID in the actual value column as "MT" and add the required values to support and deny the hypothesis.
6. Update the sub-totals and totals in the **H0** hypothesis analysis report, the **H1** hypothesis analysis report, the **H2** hypothesis analysis report, and the Measurement Testing Analysis Report.

Automation Assessment: AA

Experiment Overview

In section 4.3 the fourth claim made was that the measurement framework uses a common set of components to classify different software model types, classify different software models, measure the size of different software models, and measure the amount of reuse for different software models. This methodology phase is an experiment designed to test if this is true. The experiment can be summarised as follows:

Step 1. For each attempt at classification of a software model type in the software model type classification methodology phase, a log is made to track use of implementation components in the prototype tool.

Step 2. For each attempt at classification of a software model in the software model classification methodology phase, a log is made to track the use of implementation components in the prototype tool.

Step 3. For each attempt at measuring size of a software model in the software model classification methodology phase, a log is made to track the use of implementation components in the prototype tool.

Step 4. For each attempt at measuring the amount of reuse of a software model in the measurement testing methodology phase, a log is made to track the use of implementation components in the prototype tool.

This is the essence of the collection process.

There are five basic outcomes for any given software model type in the experiment. These outcomes are not all mutually exclusive.

Outcome 1. The number of conditional statements in a function or class method in source code that automates the measurement framework is more than or equal to the number of software model types used to test the measurement framework in the first three methodology phases, or a function or class method in source code that automates the measurement framework contains one or more calls to external programs. This supports hypothesis H2.

Outcome 2. Different meta-level components from the measurement framework are used in different test cases in the first three methodology phases (Software Model Type Classification, Software Model Classification, Measurement Testing). This supports hypothesis H2.

Outcome 3. Meta-level components were added, modified, or deleted during execution of the first three methodology phases. This includes static and dynamic meta-level components. This supports hypothesis H2.

Outcome 4. Source code that implements the measurement framework was altered during execution of the first three methodology phases. This supports hypothesis H2.

Outcome 5. The static part of the meta-model architecture was altered during execution of the first three methodology phases. This supports hypothesis H2.

Implementation components refers to source code, database tables, dialog boxes, and menu items in the prototype tool.

Based on the above outcomes, support for H2 can be quantified as follows:

- H2 is supported by the number of functions or class methods in source code that contain more conditional statements than the number of software model types used to test the measurement framework in the first three methodology phases and the number of functions or class methods in source code that contain one or more calls to external programs.
- H2 is supported by the number of times a different set of meta-level components from the measurement framework are used to execute a test case in any one of the first three methodology phases.
- H2 is supported by the number of times a meta-level component is added, deleted, or modified during execution of the first three methodology phases.
- H2 is supported by the number of times the source code that automates the measurement framework is altered during execution of the first three methodology phases.
- H2 is supported by the number of times the static part of the meta-model architecture is changed during execution of the first three methodology phases.

The five basic quantities above are the essence of hypothesis measures and assessment criteria in this methodology phase.

This methodology phase has the onerous task of trying to measure how general the measurement framework is when measuring reuse for different kinds of software models. What does this mean? how can it be measured? Before describing the experiment in more detail, the next section elaborates on the concept of a general framework with respect to the measurement framework and the automated version of the measurement framework.

Defining and Measuring the Concept of a General Framework

Defining what a general framework is can be derived from the concept of a general function. Consider a function called `TrueAdd` that can add any two numbers in C++ below.

```
int TrueAdd(int A, int B)
{
 return (A + B);
};
```

Test cases can be defined to verify that the above function can in fact add two numbers together as follows.

	Test Data		
Case #	**A**	**B**	**int TrueAdd(A,B)**
1	1	3	4
2	4	6	10
3	8	9	17
4	12	1	13

So in its simplest form it can be argued that if the same function in source code that implements some part of the measurement framework is used to classify a number of software models then the framework may be considered general, especially if the software models are instances of different software model types. Each software model type provides at least one different kind of test case.

Provided there is a mapping from the measurement framework to source code that implements the measurement framework, logging the execution of the prototype tool can indicate how general the framework is by detecting which parts of source code that map to parts of the measurement framework are used to perform a test. For example, the software model set MLC maps to a TSoftwareModelSet class. If this class is used in different test cases for measurement of size in the software model classification methodology phase then this suggests that the framework is general. Mapping tables that map parts of the measurement framework to source code were made these were described in section 3.8. This kind of analysis can be done and not only for source code, but menu items, database tables and dialog boxes because these were also mapped to parts of the measurement framework.

Indicators for determining of the framework is generally applicable are based on use of functionality in the tool can be defined as follows:

- Use of the same implementation components (functions, menu items, dialog boxes, classes and methods) that map to MLCs in the measurement framework suggest the framework is general. More test cases using the same implementation components add weight to this argument.
- Use of different implementation components (functions, menu items, dialog boxes, classes and methods) that map to MLCs in the measurement framework suggest the framework is not general. More test cases using different implementation components add weight to this argument.

So far so good, but what if someone wanted to write code that was not really general but appeared to behave as if it was general. Let us return to the function for adding two integers again. Now given the test data three kinds of different functions could be written that are not really general but behave as if they are general by generating the correct output.

To illustrate this point, three kinds of add functions are defined below in C++ that meet the test case criteria for cases 1-4, but are not considered general. These are the `IfThenAdd`, the `SwitchAdd`, and the `ExecAdd` below.

```
int IfThenAdd(int A, int B)
{
if ((A = 1) && (B = 3)) return 4;
if ((A = 4) && (B = 6)) return 10;
if ((A = 8) && (B = 9)) return 17;
if ((A = 12) && (B = 1)) return 13;
};


int SwitchAdd(int A, int B)
{
  switch (A)
  {
    case 1:switch (B)
           {
            case 3:return 4;
           }
    case 4: switch (B)
           {
            case 6:return 10;
           }
    case 8: switch (B)
           {
            case 9:return 17;
           }
    case 12: switch (B)
           {
            case 1:return 13;
           }
  };
};

int ExecAdd(int A, int B)
{
  Exec("ADD.EXE",A,B);
};
```

The `IfThenAdd` function uses different source code lines are executed for each test case so it really is not a general function. So if different values, like 0 for A, and 0 for B are passed to this function then it will most likely return the wrong value assuming the program does not halt.

The `SwitchAdd` function also uses different source code lines are executed each tests case. This function has similar consequences for other values not defined based the cases in the switch statement. For example, if values like 0 for A, and 0 for B are passed to this function then it will most likely return the wrong value assuming the program does not halt.

The `ExecAdd` function may still return the right value for values not defined in the test data, but how this is done is unknown because an external executable is used to perform the calculation. The structure of the executable's source code may contain if then statements, switch statements or both to perform the required function. So we cannot safely assume that it is general.

In the case of the measurement framework, a function may contain a switch statement that uses a different function for each kind of software model, or worse, each specific test case for classification of a software model, measurement of its size, and measurement of reuse between pairs of software models. If a new kind of software model (new test case) is used then the results are likely to be unreliable.

This leads to another indicator to determine of the measurement framework is generally applicable (A conditional test is either a case in a switch statement, a logical expression in an if-then statement, or a logical expression in a for or while loop).

- Functions or methods containing conditional tests that number more than the number of test cases for a given software model type suggest that the measurement framework is ***not generally applicable*** to measurement of the amount of reuse for different software model types.
- Functions or methods containing conditional tests that number more than the number of software model type/implementation pairs tested suggest that the measurement framework is ***not generally applicable*** to measurement of the amount of reuse for different software model types.
- Functions or methods that contain no conditional tests and no external calls suggest that the measurement framework **is generally applicable** to measurement of the amount of reuse for different software model types.
- Functions or methods that contain no external calls and also contain conditional tests that number less than the number of test cases for a given software model type/implementation pair suggests that the measurement framework is **likely to be generally applicable** to measurement of the amount of reuse for different software model types.
- Functions or methods that contain no external calls and also contain conditional tests that number less than the number of software model type/implementation pairs tested suggests that the measurement framework is **likely to be generally applicable** to measurement of the amount of reuse for different software model types.
- Functions or methods that contain external calls suggest that the measurement framework is ***not generally applicable*** to measurement of the amount of reuse for different software model types.

OK, but what if the measurement framework or the prototype tool was changed to cope with different kinds of software models? This necessarily implies a different measurement framework and/or different automated version of it was used to measure reuse for different kinds of software models. There may be few or no conditional tests in the source code but a different version is being used for each software model type. This would be like using a different version of the `SwitchAdd` function for each test case as follows.

```
int SwitchAdd(int A, int B)// for test case #1
{
  switch (A)
  {
    case 1:switch (B)
           {
            case 3:return 4;
           }
  };
};

int SwitchAdd(int A, int B)// for test case #2
{
  switch (A)
  {
    case 4: switch (B)
           {
            case 6:return 10;
           }
  };
};

int SwitchAdd(int A, int B)// for test case #3
{
  switch (A)
  {
    case 8: switch (B)
           {
            case 9:return 17;
           }
  };
};


int SwitchAdd(int A, int B)// for test case #4
{
  switch (A)
  {
    case 12: switch (B)
           {
            case 1:return 13;
           }
  };
};
```

The source code is changed for each test case and new versions are made to cope with new test cases. For example, if A is 0 and B is 0, then a new version is defined as follows.

```
int SwitchAdd(int A, int B)// for A = 0 and B = 0
{
  switch (A)
  {
    case 0: switch (B)
            {
             case 0:return 0;
            }
  };
};
```

This leads to the following indicators to determine if a the measurement framework is generally applicable:

- Changes to implementation components that automate the measurement framework suggest that the measurement framework is not generally applicable to measurement of the amount of reuse for different software model types.
- Changes to meta-level components in the static part of the meta-model architecture suggest that the measurement framework is not generally applicable to measurement of the amount of reuse for different software model types.

Although one would expect that changes to the meta-model architecture in the measurement framework would result in changes to the prototype tool, it is safer, and recommended, to track changes to both the measurement framework and the prototype tool. Now follows a description of the methodology phase in more detail.

Collection Process

Assessed Components

The assessed components in the framework for this phase are:

1. The TDL MLC at meta-level 5.
2. The TDL MLC at meta-level 4.
3. The M3 Model Specifier MLC.
4. The UML MLC at meta-level 4.
5. The Generic M3 Model MLC.
6. The UML MLC at meta-level 3.
7. The TDL MLC at meta-level 3.
8. The Set Theory MLC at meta-level 3.
9. The Measurement Model Specifier MLC.
10. The Amount of Reuse Measurement M2 Model MLC.
11. The Software Model Type Set Theory MLC.
12. The Software Model Type Classification MLC.
13. The Amount of Reuse Measurement M1 Model Specifier MLC.
14. The UML MLC at meta-level 2.
15. The TDL MLC at meta-level 2.
16. The Set Theory MLC at meta-level 2.
17. The Software Model Set Theory MLC.
18. The Software Model Classification MLC.
19. The Amount of Reuse Measurement Model Specifier MLC.
20. Measurement Data Classification MLC.

Hypotheses Tested

The hypotheses tested are:

H0b.

H2 and **SH 2.1**, **SH 2.2**, **SH 2.3**, **SH 2.4**, **SH 2.5**, **SH 2.6**, **SH 2.7**, **SH 2.8**, **SH 2.9**, **SH 2.10**, **SH 2.11**, and **SH 2.12**.

H0a: A framework based on meta-modelling cannot support measurement of the amount of reuse for any kind of software model.

H0b: A framework based on meta-modelling does not have any limitations in measurement of the amount of reuse with different kinds of software models.

H2: A framework based on meta-modelling has limitations in measurement of the amount of reuse with different kinds of software models.

SH 2.1: A framework based on meta-modelling that can classify different kinds of software models has limitations.

SH 2.2: A framework based on meta-modelling that can classify different software models based on their type has limitations.

SH 2.3: A framework based on meta-modelling that can measure size for different kinds of software models has limitations.

SH 2.4: A framework based on meta-modelling that can measure size for different software models has limitations.

SH 2.5: A framework based on meta-modelling that can measure the amount of reuse for internal composition reuse for different kinds of software models has limitations.

SH 2.6: A framework based on meta-modelling that can measure the amount of reuse based on external composition reuse for different kinds of software models has limitations.

SH 2.7: A framework based on meta-modelling that can measure the amount of reuse for internal generation reuse for different kinds of software models has limitations.

SH 2.8: A framework based on meta-modelling that can measure the amount of reuse for external generation reuse for different kinds of software models has limitations.

SH 2.9: A framework based on meta-modelling that can measure the amount of reuse for internal composition reuse for different software models has limitations.

SH 2.10: A framework based on meta-modelling that can measure the amount of reuse for external composition reuse for different software models has limitations.

SH 2.11: A framework based on meta-modelling that can measure the amount of reuse for internal generation reuse for different software models has limitations.

SH 2.12: A framework based on meta-modelling that can measure the amount of reuse for external generation reuse for different software models has limitations.

Data Specifications

Data specifications used in this methodology phase are the **Software Model Types** table, the **Implementations** table, the **Software model types and implementations** table, and a number of data specifications based on data collection instruments from other methodology phases.

The **Software Model Types** table contains the name of each software model type and also the literature sources used to identify it.

The **Implementations** table names each implementation and the vendor that developed it.

The **Software model types and Implementations** table links software model types to implementations. Each row links one software model type to an implementation.

A number of data specifications in the Automation Assessment methodology phase are described in other methodology phases as data collection instruments. These are referenced by the **Methodology Phase Process Execution** tables data collection instrument based on the **Source Data** entry in the heading. These are:

- The **Software model type classifications** table from the Software model type classification methodology phase.
- The **Software model type Test Cases** table from the Software model classification methodology phase.
- The **MT Test Cases** table from the Measurement Testing methodology phase.

See the section under the relevant heading for the methodology phase to get further details on these data specifications.

In addition, the Automation Element tables and Automation Specification Mapping tables from the Automation Specification are also used in this phase.

Data Collection Instruments

Data collected is documented using the **Function Generic Analysis** table, **Template Generic Analysis** table, **Class Generic Analysis** table, **Struct Generic Analysis** table, the **Process Log** table, the **Methodology Phase Process Execution** table, the **MLC Analysis** table, and the **MLC Change** table.

The **Function Generic Analysis** table contains the results of reviewing code for conditional tests and external calls in functions. There is one row for each function named in the functions table as part of the automation specification. Each row represents the results of reviewing code for a single function. Each row contains:

- The name of the function (**Function Name**).
- The number of conditional tests for all if-then-else statements, return statements, loop statements, and cases from switch statements contained in the **Function Name** (**#Total Conditional Tests**).
- The number of external calls to other executable programs in the **Function Name** referred (**#External Calls**).

The **Template Generic Analysis** table contains the results of reviewing code for a single template for conditional tests and external calls in each template method. There is one table for each template named in the templates, classes and structs table as part of the automation specification. Each row represents the results of reviewing code for a single method in the template. Each row contains:

- The name of the method in the template (**Template Method**).
- The number of conditional tests for all if-then-else statements, return statements, loop statements, and cases from all switch statements contained in the **Template Method** (**#Total Conditional Tests**).
- The number of external calls to other executable programs in the **Template Method** (**#External Calls**).

The **Class Generic Analysis** table contains the results of reviewing code for a single class for conditional tests and external calls in each class method. There is one table for each class named in the templates, classes and structs table as part of the automation specification. Each row represents the results of reviewing code for a single method in the class. Each row contains:

- The name of the method in the class (**Class Method**).
- The number of conditional tests for all if-then-else statements, return statements, loop statements, and cases from all switch statements contained in the **Class Method** (**#Total Conditional Tests**).
- The number of external calls to other executable programs in the **Class Method** (**#External Calls**).

The **Struct Generic Analysis** table contains the results of reviewing code for a single class for conditional tests and external calls in each struct method. There is one table for each struct named in the templates, classes and structs table as part of the automation specification. Each row represents the results of reviewing code for a single method in the struct. Each row contains:

- The name of the method in the struct (**Struct Method**).

- The number of conditional tests for all if-then-else statements, return statements, loop statements, and cases from all switch statements contained in the **Struct Method (#Total Conditional Tests)**.
- The number of external calls to other executable programs in the **Struct Method (#External Calls)**.

The **Process Log** table links a process log on a file to a unique identifier for the process log. There is one row for each process log. The process log contains a record of execution that documents the use of menu items, dialog boxes, database tables, templates, classes, structs, and functions that form part of the automation specification.

The **Methodology Phase Process Execution** table links a process log to the execution of a test case in a methodology phase. The nature of the test case is based on the data specification for the methodology phase. There are a number of tables for each methodology phase. There is one table for each software model type with an implementation used in a methodology phase and, where applicable, the approach to reuse. Each row links one test case to a process log.

The **MLC Analysis** table documents the use of an MLC based on the process log referenced in the **Methodology Phase Process Execution** tables and the mapping of the MLC to the automation specification. There is one table per MLC. Each row represents use of one MLC and its use in different test cases and different methodology phases. The row contains a number of sub-rows one for each test case in a methodology phase that the MLC is used in.

The **MLC Change** table documents changes to MLCs related to the meta-model architecture according to the order of execution of test cases for all methodology phases and its effect on the automation specification. Each row represents results of changes to MLCs' after execution of the test case identified in the row.

Collection Process Description

1. Each MLC mapped to a function, template, class, or struct in the automation specification was reviewed for its conditional tests and external calls. This is documented as one row per function using the appropriate **Generic Analysis** table.
2. One **Methodology Phase Process Execution** table was created for each methodology phase and, where applicable, each approach to reuse. These were Software Model Type Classification, Software Model Classification, and Measurement Testing.

3. Each test case for a methodology phase was re-executed and with the process of execution documented using a process log. The order of execution of methodology phases was Software Model Type Classification, Software Model Classification, and Measurement Testing.
4. The use of MLCs in the meta-model architecture via the automation specification was documented using one **MLC Analysis** table for each MLC. The effect on the meta-model architecture was documented after execution of each test case from each **Methodology Phase Process Execution** table.
5. The effect of each test case on MLCs related to the meta-model architecture for measurement and the automation specification was documented using the **MLC Change** table. Each test case was documented as one row in the **MLC Change** table.

Variables

Independent Variables

The following independent variables are obtained after execution of the collection process for the automation assessment phase.

Independent variables consist of:

1. The **Software model types** table.
2. The **Implementations** table.
3. The **Software model types and Implementations** table.
4. The MLC instances of the Amount of Reuse Measurement M2 Model MLC.
5. The MLC instances of the Software Model Type Classification MLC.
6. The MLC instances of the Software Model Type Set Theory MLC.
7. The MLC instances of the Amount of Reuse Measurement M1 Model Specifier MLC.
8. The MLC instances of the Software Model Classification MLC.
9. The MLC instances of the Software Model Set Theory MLC.
10. The MLC instances of the Amount of Reuse Measurement Model Specifier MLC.
11. The **Automation Element** tables from Automation Specification.
12. The **Mapping** tables from the automation specification.

13. All tables referenced by the **Methodology Phase Process Execution** tables via **Source Data.**
14. The **Methodology Phase Process Execution** tables.
15. The TDL MLC at meta-level 5.
16. The TDL MLC at meta-level 4.
17. The M3 Model Specifier MLC.
18. The UML MLC at meta-level 4.
19. The Generic M3 Model MLC.
20. The UML MLC at meta-level 3.
21. The TDL MLC at meta-level 3.
22. The Set Theory MLC at meta-level 3.
23. The Measurement Model Specifier MLC.
24. The Amount of Reuse Measurement M2 Model MLC.
25. The Software Model Type Set Theory MLC.
26. The Software Model Type Classification MLC.
27. The Amount of Reuse Measurement M1 Model Specifier MLC.
28. The UML MLC at meta-level 2.
29. The TDL MLC at meta-level 2.
30. The Set Theory MLC at meta-level 2.
31. The Software Model Set Theory MLC.
32. The Software Model Classification MLC.
33. The Amount of Reuse Measurement Model Specifier MLC.
34. The Measurement Data Classification MLC.

Dependent Variables

The following dependent variables are obtained after execution of the collection process for the automation assessment methodology phase.

Dependent variables consist of:

1. The **Function Generic Analysis** table, **Template Generic Analysis** tables, **Class Generic Analysis** tables, and **Struct Generic Analysis** tables.
2. The **Process Log** tables. Measures derived from this variable are the number of **Process Log** tables and the number of process logs.
3. The **MLC Analysis** tables.
4. The **MLC Change** table.

Analysis Process

Hypothesis Measures

Hypothesis measures for base indicators and limitation indicators used in this phase are tabulated in Table 4-24, Table 4-25, and Table 4-26. See Appendix C5 for a guide to interpretation of the hypothesis measures used in this methodology phase.

Table 4-24 lists the base indicators used in the automation assessment methodology phase to support calculations of limitation indicators. These include the number of test cases for each methodology phase and the number of software model types used.

Table 4-24: Base Indicators used in Automation Assessment

Model Type Category	Names of Measures
All Software Model Types	**T_CC_BI, SMTC_CC_BI, SMC_CC_BI, MT_CC_BI, MTRA_CC_BI, SMTIR_BI**
Object-oriented/UML Software Model Types	**OU_T_CC_BI, OU_SMTC_CC_BI, OU_SMC_CC_BI, OU_MT_CC_BI, OU_MTRA_CC_BI, OU_SMTIR_BI**

Below are six examples of base indicators from the guide to interpretation (Appendix C5).

Name of Measure	Calculation
T_CC_BI (Total Change Cases Base Indicator)	#rows in the **MLC Change** table
SMTC_CC_BI (SMTC Change Cases Base Indicator)	#rows in the **MLC Change** table where **MP ID** = SMTC
SMC_CC_BI (SMC Change Cases Base Indicator)	#rows in the **MLC Change** table where **MP ID** = SMC
MT_CC_BI (MT Change Cases Base Indicator)	#rows in the **MLC Change** table where **MP ID** = MT
MTRA_CC_BI (MT Reuse Approach Change Cases Base Indicator)	#rows in the **MLC Change** table where **MP ID** = MT and **Reuse Approach** = <Specific Approach>
SMTIR_BI (Software Model Type Implementation References Base Indicator)	#rows in the **Software model types and implementations** table where **SMTI ID** = **SMTI ID** in any **Methodology Phase Process Execution** table

<Specific Approach> = Internal Composition Reuse | External Composition Reuse | Internal Generation Reuse | External Generation Reuse

T_CC_BI: This indicator is the total number of test cases for all methodology phases. The value is obtained by counting the number of rows in the **MLC Change** table.

SMTC_CC_BI: This indicator is the number of test cases for the software model type classification methodology phase. The value is obtained by counting the number of rows in the **MLC Change** table for the software model type classification methodology phase.

SMC_CC_BI: This indicator is the number of test cases for the software model classification methodology phase. The value is obtained by counting the number of rows in the **MLC Change** table for the software model classification methodology phase.

MT_CC_BI: This indicator is the number of test cases for the measurement testing methodology phase. The value is obtained by counting the number of rows in the **MLC Change** table for the measurement testing methodology phase.

MTRA_CC_BI: This indicator is the number of test cases for the measurement testing methodology phase for a specific approach to reuse. There is one indicator for each approach to reuse. The value is obtained by counting the number of rows in the **MLC Change** table for the measurement testing methodology phase for a specific approach to reuse.

SMTIR_BI: This indicator is the number of software model types with implementations used in the automation assessment methodology phase. The value is obtained by counting the number of rows in the **Software model types with implementations** table that also appear in one or more **Methodology Phase Process Execution** tables via **SMTI ID**.

Table 4-25 lists the limitation indicators used in the automation assessment methodology phase to determine how general the measurement framework is based on conditional tests and use of meta-level components.

Table 4-25: Limitation Indicators used in Automation Assessment for conditional tests and use of MLCs

Model Type Category	Names of Measures
All Software Model Types	Category 1: MINCOT_LI, MAXCOT_LI, MEACOT_LI, MEDCOT_LI, MODCOT_LI, MINEC_LI, MAXEC_LI, MEAEC_LI, MEDEC_LI, MODEC_LI Category 2: MLCAEU_LI, SMIN_MLCAEU_LI, SMAX_MLCAEU_LI, SMED_MLCAEU_LI, SMEA_MLCAEU_LI, SMOD_MLCAEU_LI Category 3: SMC_MLCAEU_LI, MT_MLCAEU_LI, MT_MLCAEICU_LI, MT_MLCAEECU_LI, MT_MLCAEIGU_LI, MT_MLCAEEGU_LI Category 4: SMTC_SMIN_MLCAEU_LI, SMTC_SMIN_MLCAEU_LI, SMTC_SMED_MLCAEU_LI, SMTC_SMEA_MLCAEU_LI, SMTC_SMOD_MLCAEU_LI, SMC_SMIN_MLCAEU_LI, SMC_SMAX_MLCAEU_LI, SMC_SMED_MLCAEU_LI, SMC_SMEA_MLCAEU_LI, SMC_SMOD_MLCAEU_LI, MT_SMIN_MLCAEU_LI, MT_SMAX_MLCAEU_LI, MT_SMED_MLCAEU_LI, MT_SMEA_MLCAEU_LI, MT_SMOD_MLCAEU_LI Category 5: MT_SMIN_MLCAEICU_LI, MT_SMAX_MLCAEICU_LI, MT_SMED_MLCAEICU_LI, MT_SMEA_MLCAEICU_LI, MT_SMOD_MLCAEICU_LI, MT_SMIN_MLCAEECU_LI, MT_SMAX_MLCAEECU_LI, MT_SMED_MLCAEECU_LI, MT_SMEA_MLCAEECU_LI, MT_SMOD_MLCAEECU_LI, MT_SMIN_MLCAEIGU_LI, MT_SMAX_MLCAEIGU_LI, MT_SMED_MLCAEIGU_LI, MT_SMEA_MLCAEIGU_LI, MT_SMOD_MLCAEIGU_LI, MT_SMIN_MLCAEEGU_LI, MT_SMAX_MLCAEEGU_LI, MT_SMED_MLCAEEGU_LI, MT_SMEA_MLCAEEGU_LI, MT_SMOD_MLCAEEGU_LI,
Object-Oriented/UML Software Model Types	Category 2: OU_MLCAEU_LI, OU_SMIN_MLCAEU_LI, OU_SMAX_MLCAEU_LI, OU_SMED_MLCAEU_LI, OU_SMEA_MLCAEU_LI, OU_SMOD_MLCAEU_LI Category 3: OU_SMC_MLCAEU_LI, OU_MT_MLCAEU_LI, OU_MT_MLCAEICU_LI, OU_MT_MLCAEECU_LI, OU_MT_MLCAEIGU_LI, OU_MT_MLCAEEGU_LI Category 4: OU_SMTC_SMIN_MLCAEU_LI, OU_SMTC_SMAX_MLCAEU_LI, OU_SMTC_SMED_MLCAEU_LI, OU_SMTC_SMEA_MLCAEU_LI, OU_SMTC_SMOD_MLCAEU_LI, OU_SMC_SMIN_MLCAEU_LI, OU_SMC_SMAX_MLCAEU_LI, OU_SMC_SMED_MLCAEU_LI, OU_SMC_SMEA_MLCAEU_LI, OU_SMC_SMOD_MLCAEU_LI, OU_MT_SMIN_MLCAEU_LI, OU_MT_SMAX_MLCAEU_LI, OU_MT_SMED_MLCAEU_LI, OU_MT_SMEA_MLCAEU_LI, OU_MT_SMOD_MLCAEU_LI Category 5: OU_MT_SMIN_MLCAEICU_LI, OU_MT_SMAX_MLCAEICU_LI, OU_MT_SMED_MLCAEICU_LI, OU_MT_SMEA_MLCAEICU_LI, OU_MT_SMOD_MLCAEICU_LI, OU_MT_SMIN_MLCAEECU_LI, OU_MT_SMAX_MLCAEECU_LI, OU_MT_SMED_MLCAEECU_LI, OU_MT_SMEA_MLCAEECU_LI, OU_MT_SMOD_MLCAEECU_LI, OU_MT_SMIN_MLCAEIGU_LI, OU_MT_SMAX_MLCAEIGU_LI, OU_MT_SMED_MLCAEIGU_LI, OU_MT_SMEA_MLCAEIGU_LI, OU_MT_SMOD_MLCAEIGU_LI, OU_MT_SMIN_MLCAEEGU_LI, OU_MT_SMAX_MLCAEEGU_LI, OU_MT_SMED_MLCAEEGU_LI, OU_MT_SMEA_MLCAEEGU_LI, OU_MT_SMOD_MLCAEEGU_LI

Category 1 indicators measure the number of conditional statements and external calls in the source code that implements the measurement framework (No separate indicator used for object-oriented and UML software model types).

Category 2 indicators measure the level of use of MLCs across all methodology phases in total and for specific software model types.

Category 3 indicators measure the level of use of MLCs for specific methodology phases and specific approaches to reuse.

Category 4 indicators measure the level of use of MLCs for specific methodology phases and specific software model types.

Category 5 indicators measure the level of use of MLCs for specific software model types and specific approaches to reuse in the measurement testing methodology phase.

Below are six examples of limitation indicators based from the guide to interpretation (Appendix C5). The first and second examples are category one indicators that quantify outcome 1 from the experiment overview. The third example is a category 2 indicator that quantifies outcome 2 from the experiment overview. The fourth example is a category 3 indicator that quantifies outcome 2 from the experiment overview. The fifth example is a category 4 indicator that quantifies outcome 2 from the experiment overview. The sixth example is a category 5 indicator that quantifies outcome 2 from the experiment overview.

Name of Measure	Calculation
MAXCOT_LI (Maximum Conditional Tests Limitation Indicator)	(highest value for **#Total Conditional Tests** in all <Specific> **Generic Analysis** tables) ÷ (**SMTIR_BI**)
MAXEC_LI (Maximum External Calls Limitation Indicator)	(highest value for **#External Calls** in all <Specific> **Generic Analysis** tables) ÷ (**SMTIR_BI**)
MLCAEU_LI (MLC AE Usage Limitation Indicator)	(#rows in all **MLC Analysis** tables where **MLC Name** = <Specific MLC Name>) ÷ (#rows in all **Methodology Phase Process Execution** tables)
SMC_MLCAEU_LI (SMC MLC AE Usage Limitation Indicator)	(#rows in all **MLC Analysis** tables where **MLC Name** = <Specific MLC Name> and **MP ID** = SMC) ÷ (#rows in all **Methodology Phase Process Execution** tables where **MP ID** = SMC)
SMTC_SMEA_MLCAEU_LI (SMTC Specific Software Model Type Mean MLC AE Usage Limitation Indicator)	Mean value for (#rows in all **MLC Analysis** tables where **MLC Name** = <Specific MLC Name> and **SMTI ID** = <Specific SMTI ID> and **MP ID** = SMTC) ÷ (#rows in all **Methodology Phase Process Execution** tables where **SMTI ID** = <Specific SMTI ID> and **MP ID** = SMTC)
MT_SMAX_MLCAEICU_LI (MT Specific Software Model Type Maximum MLC AE Internal Composition Usage Limitation Indicator)	Highest value for (#rows in all **MLC Analysis** tables where **MLC Name** = <Specific MLC Name> and **MP ID** = MT and **SMTI ID** = <Specific SMTI ID> and **Reuse Approach** = Internal Composition Reuse) ÷ (#rows in all **Methodology Phase Process Execution** tables where **MP ID** = MT and **SMTI ID** = <Specific SMTI ID> and **Reuse Approach** = Internal Composition Reuse)

<Specific> ::= **Function** | **Class** | **Template** | **Struct**
<Specific MLC Name>::= **MLC Name** from **Assessed Components** in the **Automation Assessment** methodology phase.
<Specific SMTI ID> ::= **SMTI ID** from the **Software Model Types and Implementations** table.

MAXCOT_LI: This indicator attempts to assess the lowest possible level of flexibility based on conditional tests in if-then-else statements and case statements of code. The value is obtained by finding the maximum value for number of if-then-else conditional tests and cases in all functions, classes, templates, or structs and dividing the result by the number of software model types with implementations used. This

indicator is compared with SMTIR_BI. A low value indicates that the automation specification is more flexible. The lowest possible value for MAXCOT_LI is zero. A high value indicates that the automation specification is less flexible. If MAXCOT_LI is less than one then this indicates that the automation specification is flexible. If MAXCOT_LI is more than one then this indicates that the automation specification is not flexible. If MAXCOT_LI equals one then this indicates that the automation specification is borderline between flexible and not flexible.

MAXEC_LI: This indicator attempts to assess the lowest possible level of flexibility based on external calls in statements of code. The value is obtained by finding the maximum value for number of external calls in all functions, classes, templates, or structs and dividing the result by the number of software model types with implementations used. This indicator is compared with SMTIR_BI. A low value indicates that the automation specification is more flexible. The lowest possible value for MAXEC_LI is zero. A high value indicates that the automation specification is less flexible. If MAXEC_LI is less than one then this indicates that the automation specification is flexible. If MAXEC_LI is more than or equal to one then this indicates that the automation specification is not flexible.

MLCAEU_LI: This indicator attempts to assess the flexibility of the meta-model architecture based on use of MLCs via their mapping to automation elements in the automation specification. There is one indicator for each MLC mapped to an automation element. The value is obtained by finding all instances where an MLC is used based on its mapping to an automation element and dividing the result by all test cases for all methodology phases. A high value indicates a high level of flexibility for the MLC. The highest possible value for MLCAEU_LI is one. A low value indicates a low level of flexibility for the MLC. The lowest possible value for MLCAEU_LI is zero.

SMC_MLCAEU_LI: This indicator attempts to assess the flexibility of the meta-model architecture based on use of MLCs via their mapping to automation elements in the automation specification in the software model classification methodology phase. There is one indicator for each MLC mapped to an automation element. The value is obtained by finding all instances where an MLC is used based on its mapping to an automation element in the software model classification methodology phase and dividing the result by all test cases in the software model classification methodology phase. A high value indicates a high level of flexibility for the MLC. The highest possible value for SMC_MLCAEU_LI is one. A low value indicates a low level of flexibility for the MLC. The lowest possible value for SMC_MLCAEU_LI is zero.

SMTC_SMEA_MLCAEU_LI: This indicator attempts to assess the flexibility of the meta-model architecture based on use of MLCs via their mapping to automation elements in the automation specification for specific software model types in the software model type classification methodology phase. There is one indicator for each MLC mapped to an automation element. Based on an MLC's mapping to automation elements, the value is obtained by finding the mean value for all instances where the MLC is used in the software model type classification methodology phase for a given software model type and implementation referenced and dividing the result by all test cases in the software model type classification methodology phase for that software model type and implementation. A high value indicates a high level of flexibility for the MLC. The highest possible value for SMTC_SMEA_MLCAEU_LI is one. A low value indicates a low level of flexibility for the MLC. The lowest possible value for SMTC_SMEA_MLCAEU_LI is zero.

MT_SMAX_MLCAEICU_LI: This indicator attempts to assess the flexibility of the meta-model architecture based on use of MLCs via their mapping to automation elements in the automation specification for specific software model types in the measurement testing methodology phase for the internal composition reuse approach. There is one indicator for each MLC mapped to an automation element. Based on an MLC's mapping to automation elements, the value is obtained by finding the highest value for all instances where the MLC is used in the measurement testing methodology phase for internal composition reuse with a given software model type and implementation and dividing the result by all test cases in the measurement testing methodology phase for internal composition reuse with that software model type and implementation. A high value indicates a high level of flexibility for the MLC. The highest possible value for MT_SMAX_MLCAEICU_LI is one. A low value indicates a low level of flexibility for the MLC. The lowest possible value for MT_SMAX_MLCAEICU_LI is zero.

Table 4-26 lists the limitation indicators used in the automation assessment methodology phase for assessment of changes in MLCs.

Table 4-26: Limitation Indicators to measure changes to the measurement framework

Model Type Category	Names of Measures
All Software Model Types	**Category 1: T_MLCR_LI, T_MLCNR_LI, T_MLCA_LI, T_MLCNA_LI, T_MLCM_LI, T_MLCNM_LI, SMTC_MLCR_LI, SMTC_MLCNR_LI, SMTC_MLCA_LI, SMTC_MLCNA_LI, SMTC_MLCM_LI, SMTC_MLCNM_LI, SMC_MLCR_LI, SMC_MLCNR_LI, SMC_MLCA_LI, SMC_MLCNA_LI, SMC_MLCM_LI, SMC_MLCNM_LI, MT_MLCR_LI, MT_MLCNR_LI, MT_MLCA_LI, MT_MLCNA_LI, MT_MLCM_LI, MT_MLCNM_LI, MTRA_MLCR_LI, MTRA_MLCNR_LI, MTRA_MLCA_LI, MTRA_MLCNA_LI, MTRA_MLCM_LI, MTRA_MLCNM_LI** **Category 2: T_ASPEC_LI, T_ASPTC_LI, SMTC_ASPEC_LI, SMTC_ASPTC_LI, SMC_ASPEC_LI, SMC_ASPTC_LI, MT_ASPEC_LI, MT_ASPTC_LI, MTRA_ASPEC_LI, MTRA_ASPTC_LI** **Category 3: T_MMAPEC_LI, T_MMAPTC_LI, SMTC_MMAPEC_LI, SMTC_MMAPTC_LI, SMC_MMAPEC_LI, SMC_MMAPTC_LI, MT_MMAPEC_LI, MT_MMAPTC_LI, MTRA_MMAPEC_LI, MTRA_MMAPTC_LI**
Object-oriented/UML Software Model Types	**Category 1: OU_T_MLCR_LI, OU_T_MLCNR_LI, OU_T_MLCA_LI, OU_T_MLCNA_LI, OU_T_MLCM_LI, OU_T_MLCNM_LI, OU_SMTC_MLCR_LI, OU_SMTC_MLCNR_LI, OU_SMTC_MLCA_LI, OU_SMTC_MLCNA_LI, OU_SMTC_MLCM_LI, OU_SMTC_MLCNM_LI, OU_SMC_MLCR_LI, OU_SMC_MLCNR_LI, OU_SMC_MLCA_LI, OU_SMC_MLCNA_LI, OU_SMC_MLCM_LI, OU_SMC_MLCNM_LI, OU_MT_MLCR_LI, OU_MT_MLCNR_LI, OU_MT_MLCA_LI, OU_MT_MLCNA_LI, OU_MT_MLCM_LI, OU_MT_MLCNM_LI, OU_MT_RAMLCR_LI, OU_MT_RAMLCNR_LI, OU_MT_RAMLCA_LI, OU_MT_RAMLCNA_LI, OU_MT_RAMLCM_LI, OU_MT_RAMLCNM_LI** **Category 2: OU_T_ASPEC_LI, OU_T_ASPTC_LI, OU_SMTC_ASPEC_LI, OU_SMTC_ASPTC_LI, , OU_SMC_ASPEC_LI, OU_SMC_ASPTC_LI, OU_MT_ASPEC_LI, OU_MT_ASPTC_LI, OU_MT_RAASPEC_LI, OU_MT_RAASPTC_LI** **Category 3: OU_T_MMAPEC_LI, OU_T_MMAPTC_LI, OU_SMTC_MMAPEC_LI, OU_SMTC_MMAPTC_LI, OU_SMC_MMAPEC_LI, OU_SMC_MMAPTC_LI, OU_MT_MMAPEC_LI, OU_MT_MMAPTC_LI, OU_MT_RAMMAPEC_LI, OU_MT_RAMMAPTC_LI**

Category 1 indicators measure addition, modification, and removal of MLCs in the meta-model architecture across all methodology phases and for specific methodology phases.

Category 2 indicators measure changes to the source code of the automation specification across all methodology phases, specific methodology phases, and specific approaches to reuse.

Category 3 indicators measure the changes to the static part of the meta-model architecture across all methodology phases, specific methodology phases, and specific approaches to reuse.

Below are five examples of limitation indicators from the guide to interpretation (Appendix C5). The first and second examples are category 1 indicators that quantify outcome 3 from the experiment overview. The third and fourth examples are category 2 indicators that quantify outcome 4 from the experiment overview. The fifth example is a category 3 indicator that quantifies outcome 5 from the experiment overview.

Name of Measure	Calculation
T_MLCA_LI (Total MLC Added Limitation Indicator)	#rows in the **MLC Change** table where **MLC Added** > 0
SMTC_MLCR_LI (SMTC MLC Removed Limitation Indicator)	#rows in the **MLC Change** table where **MLC Removed** > 0 and **MP ID** = SMTC
T_ASPEC_LI (Total Auto Spec Pre-Change Limitation Indicator)	#rows in the **MLC Change** table where **Auto Spec Pre-Change** = Yes
SMC_ASPEC_LI (SMC Auto Spec Pre-Change Limitation Indicator)	#rows in the **MLC Change** table where **Auto Spec Pre-Change** = Yes and **MP ID** = SMC
MTRA_MMAPTC_LI (MT Reuse Approach MMA Post-Change Limitation Indicator)	#rows in the **MLC Change** table where **MMA Post-Change** = Yes and **MP ID** = MT and **Reuse Approach** = <Specific Approach>

<Specific Approach> = Internal Composition Reuse | External Composition Reuse | Internal Generation Reuse | External Generation Reuse

T_MLCA_LI: This indicator is used to assess flexibility of the meta-model architecture based on addition of MLCs from the meta-model architecture. The value is obtained by counting the number of times one or more MLCs were added for any given test case. This value is compared with TCC_BI. A low value indicates a high level of flexibility for the meta-model architecture. The lowest possible value for

T_MLCA_LI is zero. A high value indicates a low level of flexibility for the meta-model architecture. The highest possible value for T_MLCA_LI is TCC_BI.

SMTC_MLCR_LI: This indicator is used to assess flexibility of the meta-model architecture based on removal of MLCs from the meta-model architecture in the software model type classification methodology phase. The value is obtained by counting the number of times one or more MLCs were removed for any given test case in the software model type classification methodology phase. This value is compared with SMTC_CC_BI. A low value indicates a high level of flexibility for the meta-model architecture. The lowest possible value for SMTC_MLCR_LI is zero. A high value indicates a low level of flexibility for the meta-model architecture. The highest possible value for SMTC_MLCR_LI is SMTC_CC_BI.

T_ASPEC_LI: This indicator is used to assess flexibility of the meta-model architecture based on changes to the automation specification prior to execution of test cases. The value is obtained by counting the number of times the automation specification was changed prior to execution of a given test case. This value is compared with TCC_BI. A low value indicates a high level of flexibility for the meta-model architecture. The lowest possible value for T_ASPEC_LI is zero. A high value indicates a low level of flexibility for the meta-model architecture. The highest possible value for T_ASPEC_LI is TCC_BI.

SMC_ASPEC_LI: This indicator is used to assess flexibility of the meta-model architecture based on changes to the automation specification prior to execution of test cases in the software model classification methodology phase. The value is obtained by counting the number of times the automation specification was changed prior to execution of a given test case in the software model classification methodology phase. This value is compared with SMC_CC_BI. A low value indicates a high level of flexibility for the meta-model architecture. The lowest possible value for SMC_ASPEC_LI is zero. A high value indicates a low level of flexibility for the meta-model architecture. The highest possible value for SMC_ASPEC_LI is SMC_CC_BI.

MTRA_MMAPTC_LI: This indicator is used to assess flexibility of the meta-model architecture based on changes to the static part of the meta-model architecture after execution of test cases for a specific approach to reuse in the measurement testing methodology phase. There is one indicator and one base indicator for comparison with each approach to reuse. The value is obtained by counting the number of times the static part of the meta-model architecture was changed after execution of a given test case for a specific approach to reuse in the measurement testing methodology phase. This value is compared with the MTRA_CC_BI for the

specific approach to reuse. A low value indicates a high level of flexibility for the meta-model architecture. The lowest possible value for MTRA_MMAPTC_LI is zero. A high value indicates a low level of flexibility for the meta-model architecture. The highest possible value for MTRA_MMAPTC_LI is MTRA_CC_BI.

Assessment Criteria

Use of the hypothesis measures to test the hypotheses is illustrated in:

- Table 4-27 for **H0b**.
- Table 4-28 and Table 4-29 for **H2** via **SH 2.1**, **SH 2.2**, **SH 2.3**, **SH 2.4**, **SH 2.5**, **SH 2.6**, **SH 2.7**, **SH 2.8**, **SH 2.9**, **SH 2.10**, **SH 2.11**, and **SH 2.12**.

Refer to Appendix C5 for further details on descriptions for assessment criteria.

Table 4-27 names the hypothesis measures used for testing the null hypothesis **H0**, part **(b)**.

Table 4-27: Assessment of **H0b** using hypothesis measures.

Hypothesis Part	Measures Used
H0b	MINCOT_LI, MAXCOT_LI, MEACOT_LI, MEDCOT_LI, MODCOT_LI, MINEC_LI, MAXEC_LI, MEAEC_LI, MEDEC_LI, MODEC_LI, MLCAEU_LI, SMIN_MLCAEU_LI, SMAX_MLCAEU_LI, SMED_MLCAEU_LI, SMEA_MLCAEU_LI, SMOD_MLCAEU_LI, T_MLCR_LI, T_MLCNR_LI, T_MLCA_LI, T_MLCNA_LI, T_MLCM_LI, T_MLCNM_LI, T_ASPEC_LI, T_ASPTC_LI, T_MMAPEC_LI, T_MMAPTC_LI OU_MLCAEU_LI, OU_SMIN_MLCAEU_LI, OU_SMAX_MLCAEU_LI, OU_SMED_MLCAEU_LI, OU_SMEA_MLCAEU_LI, OU_SMOD_MLCAEU_LI, OU_T_MLCR_LI, OU_T_MLCNR_LI, OU_T_MLCA_LI, OU_T_MLCNA_LI, OU_T_MLCM_LI, OU_T_MLCNM_LI, OU_T_ASPEC_LI, OU_T_ASPTC_LI, OU_T_MMAPEC_LI, OU_T_MMAPTC_LI

How values for the hypothesis measures support H0 is detailed in the guide to interpretation (Appendix C5). Below are some examples from the guide to interpretation that illustrate this.

The truth of **H0b** is *supported* if:

- The highest number of conditional tests in any function, class method, struct method, or template method equals zero (MAXCOT_LI = 0).
- The highest number of external calls in any function, class method, struct method, or template method equals zero (MAXEC_LI = 0).
- All specific MLCs mapped to an Automation Element of a given type are used in all methodology phases (MLCAEU_LI = 1).
- No test cases resulted in addition of MLCs (T_MLCA_LI = 0).
- No test cases resulted in modification of the automation specification prior to execution of any given test case (T_ASPEC_LI = 0).

To *deny* **H0b**:

- The highest number of conditional tests in any function, class method, struct method, or template method must be more than zero (MAXCOT_LI > 0).
- The highest number of external calls in any function, class method, struct method, or template method must be more than zero (MAXEC_LI > 0).
- Some specific MLCs mapped to an Automation Element of a given type are used in one or more methodology phases (MLCAEU_LI < 1).
- Some test cases resulted in addition of MLCs (T_MLCA_LI > 0).
- Some test cases resulted in modification of the automation specification prior to execution of any given test case (T_ASPEC_LI > 0).

Table 4-28 and Table 4-29 name the hypothesis measures used for testing hypothesis **H2**.

Table 4-28: Assessment of Hypothesis **H2** using Hypothesis measures

Sub-Hypothesis	Measures Used
(All)	MINCOT_LI, MAXCOT_LI, MEACOT_LI, MEDCOT_LI, MODCOT_LI, MINEC_LI, MAXEC_LI, MEAEC_LI, MEDEC_LI, MODEC_LI, MLCAEU_LI, SMIN_MLCAEU_LI, SMAX_MLCAEU_LI, SMED_MLCAEU_LI, SMEA_MLCAEU_LI, SMOD_MLCAEU_LI, T_MLCR_LI, T_MLCNR_LI, T_MLCA_LI, T_MLCNA_LI, T_MLCM_LI, T_MLCNM_LI, T_ASPEC_LI, T_ASPTC_LI, T_MMAPEC_LI, T_MMAPTC_LI
SH 2.1	SMTC_SMIN_MLCAEU_LI, SMTC_SMAX_MLCAEU_LI, SMTC_SMED_MLCAEU_LI, SMTC_SMEA_MLCAEU_LI, SMTC_SMOD_MLCAEU_LI, SMTC_MLCR_LI, SMTC_MLCNR_LI, SMTC_MLCA_LI, SMTC_MLCNA_LI, SMTC_MLCM_LI, SMTC_MLCNM_LI, SMTC_ASPEC_LI, SMTC_ASPTC_LI, SMTC_MMAPEC_LI, SMTC_MMAPTC_LI
SH 2.2	SMC_MLCAEU_LI, SMC_MLCR_LI, SMC_MLCNR_LI, SMC_MLCA_LI, SMC_MLCNA_LI, SMC_MLCM_LI, SMC_MLCNM_LI, SMC_ASPEC_LI, SMC_ASPTC_LI, SMC_MMAPEC_LI, SMC_MMAPTC_LI
SH 2.3	SMC_SMIN_MLCAEU_LI, SMC_SMAX_MLCAEU_LI, SMC_SMED_MLCAEU_LI, SMC_SMEA_MLCAEU_LI, SMC_SMOD_MLCAEU_LI, SMC_MLCR_LI, SMC_MLCNR_LI, SMC_MLCA_LI, SMC_MLCNA_LI, SMC_MLCM_LI, SMC_MLCNM_LI, SMC_ASPEC_LI, SMC_ASPTC_LI, SMC_MMAPEC_LI, SMC_MMAPTC_LI
SH 2.4	SMC_MLCAEU_LI, SMC_MLCR_LI, SMC_MLCNR_LI, SMC_MLCA_LI, SMC_MLCNA_LI, SMC_MLCM_LI, SMC_MLCNM_LI, SMC_ASPEC_LI, SMC_ASPTC_LI, SMC_MMAPEC_LI, SMC_MMAPTC_LI
SH 2.5	MT_SMIN_MLCAEU_LI, MT_SMAX_MLCAEU_LI, MT_SMED_MLCAEU_LI, MT_SMEA_MLCAEU_LI, MT_SMOD_MLCAEU_LI, MT_SMIN_MLCAEICU_LI, MT_SMAX_MLCAEICU_LI, MT_SMED_MLCAEICU_LI, MT_SMEA_MLCAEICU_LI, MT_SMOD_MLCAEICU_LI, MT_MLCR_LI, MT_MLCNR_LI, MT_MLCA_LI, MT_MLCNA_LI, MT_MLCM_LI, MT_MLCNM_LI, MT_ASPEC_LI, MT_ASPTC_LI, MT_MMAPEC_LI, MT_MMAPTC_LI, MTRA_MLCR_LI, MTRA_MLCNR_LI, MTRA_MLCA_LI, MTRA_MLCNA_LI, MTRA_MLCM_LI, MTRA_MLCNM_LI, MTRA_ASPEC_LI, MTRA_ASPTC_LI, MTRA_MMAPEC_LI, MTRA_MMAPTC_LI, <Specific Approach>= Internal Composition Reuse
SH 2.6	MT_SMIN_MLCAEU_LI, MT_SMAX_MLCAEU_LI, MT_SMED_MLCAEU_LI, MT_SMEA_MLCAEU_LI, MT_SMOD_MLCAEU_LI, MT_SMIN_MLCAEECU_LI, MT_SMAX_MLCAEECU_LI, MT_SMED_MLCAEECU_LI, MT_SMEA_MLCAEECU_LI, MT_SMOD_MLCAEECU_LI, MT_MLCR_LI, MT_MLCNR_LI, MT_MLCA_LI, MT_MLCNA_LI, MT_MLCM_LI, MT_MLCNM_LI, MT_ASPEC_LI, MT_ASPTC_LI, MT_MMAPEC_LI, MT_MMAPTC_LI, MTRA_MLCR_LI, MTRA_MLCNR_LI, MTRA_MLCA_LI, MTRA_MLCNA_LI, MTRA_MLCM_LI, MTRA_MLCNM_LI, MTRA_ASPEC_LI, MTRA_ASPTC_LI, MTRA_MMAPEC_LI, MTRA_MMAPTC_LI, <Specific Approach>= External Composition Reuse

Sub-Hypothesis	Measures Used
SH 2.7	MT_SMIN_MLCAEU_LI, MT_SMAX_MLCAEU_LI, MT_SMED_MLCAEU_LI, MT_SMEA_MLCAEU_LI, MT_SMOD_MLCAEU_LI, MT_SMIN_MLCAEIGU_LI, MT_SMAX_MLCAEIGU_LI, MT_SMED_MLCAEIGU_LI, MT_SMEA_MLCAEIGU_LI, MT_SMOD_MLCAEIGU_LI, MTMLCR_LI, MT_MLCNR_LI, MT_MLCA_LI, MT_MLCNA_LI, MT_MLCM_LI, MT_MLCNM_LI, MT_ASPEC_LI, MT_ASPTC_LI, MT_MMAPEC_LI, MT_MMAPTC_LI, MTRA_MLCR_LI, MTRA_MLCNR_LI, MTRA_MLCA_LI, MTRA_MLCNA_LI, MTRA_MLCM_LI, MTRA_MLCNM_LI, MTRA_ASPEC_LI, MTRA_ASPTC_LI, MTRA_MMAPEC_LI, MTRA_MMAPTC_LI, <Specific Approach>= Internal Generation Reuse
SH 2.8	MT_SMIN_MLCAEU_LI, MT_SMAX_MLCAEU_LI, MT_SMED_MLCAEU_LI, MT_SMEA_MLCAEU_LI, MT_SMOD_MLCAEU_LI, MT_SMIN_MLCAEEGU_LI, MT_SMAX_MLCAEEGU_LI, MT_SMED_MLCAEEGU_LI, MT_SMEA_MLCAEEGU_LI, MT_SMOD_MLCAEEGU_LI, MT_MLCR_LI, MT_MLCNR_LI, MT_MLCA_LI, MT_MLCNA_LI, MT_MLCM_LI, MT_MLCNM_LI, MT_ASPEC_LI, MT_ASPTC_LI, MT_MMAPEC_LI, MT_MMAPTC_LI, MTRA_MLCR_LI, MTRA_MLCNR_LI, MTRA_MLCA_LI, MTRA_MLCNA_LI, MTRA_MLCM_LI, MTRA_MLCNM_LI, MTRA_ASPEC_LI, MTRA_ASPTC_LI, MTRA_MMAPEC_LI, MTRA_MMAPTC_LI, <Specific Approach>= External Generation Reuse
SH 2.9	MT_MLCAEU_LI, MT_MLCAEICU_LI, MT_MLCR_LI, MT_MLCNR_LI, MT_MLCA_LI, MT_MLCNA_LI, MT_MLCM_LI, MT_MLCNM_LI, MT_ASPEC_LI, MT_ASPTC_LI, MT_MMAPEC_LI, MT_MMAPTC_LI, MTRA_MLCR_LI, MTRA_MLCNR_LI, MTRA_MLCA_LI, MTRA_MLCNA_LI, MTRA_MLCM_LI, MTRA_MLCNM_LI, MTRA_ASPEC_LI, MTRA_ASPTC_LI, MTRA_MMAPEC_LI, MTRA_MMAPTC_LI, <Specific Approach>= Internal Composition Reuse
SH 2.10	MT_MLCAEU_LI, MT_MLCAEECU_LI, MT_MLCR_LI, MT_MLCNR_LI, MT_MLCA_LI, MT_MLCNA_LI, MT_MLCM_LI, MT_MLCNM_LI, MT_ASPEC_LI, MT_ASPTC_LI, MT_MMAPEC_LI, MT_MMAPTC_LI, MTRA_MLCR_LI, MTRA_MLCNR_LI, MTRA_MLCA_LI, MTRA_MLCNA_LI, MTRA_MLCM_LI, MTRA_MLCNM_LI, MTRA_ASPEC_LI, MTRA_ASPTC_LI, MTRA_MMAPEC_LI, MTRA_MMAPTC_LI, <Specific Approach>= External Composition Reuse
SH 2.11	MT_MLCAEU_LI, MT_MLCAEIGU_LI, MT_MLCR_LI, MT_MLCNR_LI, MT_MLCA_LI, MT_MLCNA_LI, MT_MLCM_LI, MT_MLCNM_LI, MT_ASPEC_LI, MT_ASPTC_LI, MT_MMAPEC_LI, MT_MMAPTC_LI, MTRA_MLCR_LI, MTRA_MLCNR_LI, MTRA_MLCA_LI, MTRA_MLCNA_LI, MTRA_MLCM_LI, MTRA_MLCNM_LI, MTRA_ASPEC_LI, MTRA_ASPTC_LI, MTRA_MMAPEC_LI, MTRA_MMAPTC_LI, <Specific Approach>= Internal Generation Reuse
SH 2.12	MT_MLCAEU_LI, MT_MLCAEEGU_LI, MT_MLCR_LI, MT_MLCNR_LI, MT_MLCA_LI, MT_MLCNA_LI, MT_MLCM_LI, MT_MLCNM_LI, MT_ASPEC_LI, MT_ASPTC_LI, MT_MMAPEC_LI, MT_MMAPTC_LI, MTRA_MLCR_LI, MTRA_MLCNR_LI, MTRA_MLCA_LI, MTRA_MLCNA_LI, MTRA_MLCM_LI, MTRA_MLCNM_LI, MTRA_ASPEC_LI, MTRA_ASPTC_LI, MTRA_MMAPEC_LI, MTRA_MMAPTC_LI, <Specific Approach>= External Generation Reuse

Table 4-29: Assessment of Hypothesis **H2** using Hypothesis measures

Sub-Hypothesis	Measures Used
(All)	OU_MLCAEU_LI, OU_SMIN_MLCAEU_LI, OU_SMAX_MLCAEU_LI, OU_SMED_MLCAEU_LI, OU_SMEA_MLCAEU_LI, OU_SMOD_MLCAEU_LI, OU_T_MLCR_LI, OU_T_MLCNR_LI, OU_T_MLCA_LI, OU_T_MLCNA_LI, OU_T_MLCM_LI, OU_T_MLCNM_LI, OU_T_ASPEC_LI, OU_T_ASPTC_LI, OU_T_MMAPEC_LI, OU_T_MMAPTC_LI
SH 2.1	OU_SMTC_SMIN_MLCAEU_LI, OU_SMTC_SMAX_MLCAEU_LI, OU_SMTC_SMED_MLCAEU_LI, OU_SMTC_SMEA_MLCAEU_LI, OU_SMTC_SMOD_MLCAEU_LI, OU_SMTC_MLCR_LI, OU_SMTC_MLCNR_LI, OU_SMTC_MLCA_LI, OU_SMTC_MLCNA_LI, OU_SMTC_MLCM_LI, OU_SMTC_MLCNM_LI, OU_SMTC_ASPEC_LI, OU_SMTC_ASPTC_LI, OU_SMTC_MMAPEC_LI, OU_SMTC_MMAPTC_LI
SH 2.2	OU_SMC_MLCAEU_LI, OU_SMC_MLCR_LI, OU_SMC_MLCNR_LI, OU_SMC_MLCA_LI, OU_SMC_MLCNA_LI, OU_SMC_MLCM_LI, OU_SMC_MLCNM_LI, OU_SMC_ASPEC_LI, OU_SMC_ASPTC_LI, OU_SMC_MMAPEC_LI, OU_SMC_MMAPTC_LI
SH 2.3	OU_SMC_SMIN_MLCAEU_LI, OU_SMC_SMAX_MLCAEU_LI, OU_SMC_SMED_MLCAEU_LI, OU_SMC_SMEA_MLCAEU_LI, OU_SMC_SMOD_MLCAEU_LI, OU_SMC_MLCR_LI, OU_SMC_MLCNR_LI, OU_SMC_MLCA_LI, OU_SMC_MLCNA_LI, OU_SMC_MLCM_LI, OU_SMC_MLCNM_LI, OU_SMC_ASPEC_LI, OU_SMC_ASPTC_LI, OU_SMC_MMAPEC_LI, OU_SMC_MMAPTC_LI
SH 2.4	OU_SMC_MLCAEU_LI, OU_SMC_MLCR_LI, OU_SMC_MLCNR_LI, OU_SMC_MLCA_LI, OU_SMC_MLCNA_LI, OU_SMC_MLCM_LI, OU_SMC_MLCNM_LI, OU_SMC_ASPEC_LI, OU_SMC_ASPTC_LI, OU_SMC_MMAPEC_LI, OU_SMC_MMAPTC_LI
SH 2.5	OU_MT_SMIN_MLCAEU_LI, OU_MT_SMAX_MLCAEU_LI, OU_MT_SMED_MLCAEU_LI, OU_MT_SMEA_MLCAEU_LI, OU_MT_SMOD_MLCAEU_LI, OU_MT_SMIN_MLCAEICU_LI, OU_MT_SMAX_MLCAEICU_LI, OU_MT_SMED_MLCAEICU_LI, OU_MT_SMEA_MLCAEICU_LI, OU_MT_SMOD_MLCAEICU_LI, OU_MT_MLCR_LI, OU_MT_MLCNR_LI, OU_MT_MLCA_LI, OU_MT_MLCNA_LI, OU_MT_MLCM_LI, OU_MT_MLCNM_LI, OU_MT_ASPEC_LI, OU_MT_ASPTC_LI, OU_MT_MMAPEC_LI, OU_MT_MMAPTC_LI, OU_MTRA_MLCR_LI, OU_MTRA_MLCNR_LI, OU_MTRA_MLCA_LI, OU_MTRA_MLCNA_LI, OU_MTRA_MLCM_LI, OU_MTRA_MLCNM_LI, OU_MTRA_ASPEC_LI, OU_MTRA_ASPTC_LI, OU_MTRA_MMAPEC_LI, OU_MTRA_MMAPTC_LI, <Specific Approach>= Internal Composition Reuse
SH 2.6	OU_MT_SMIN_MLCAEU_LI, OU_MT_SMAX_MLCAEU_LI, OU_MT_SMED_MLCAEU_LI, OU_MT_SMEA_MLCAEU_LI, OU_MT_SMOD_MLCAEU_LI, OU_MT_SMIN_MLCAEECU_LI, OU_MT_SMAX_MLCAEECU_LI, OU_MT_SMED_MLCAEECU_LI, OU_MT_SMEA_MLCAEECU_LI, OU_MT_SMOD_MLCAEECU_LI, OU_MT_MLCR_LI, OU_MT_MLCNR_LI, OU_MT_MLCA_LI, OU_MT_MLCNA_LI, OU_MT_MLCM_LI, OU_MT_MLCNM_LI, OU_MT_ASPEC_LI, OU_MT_ASPTC_LI, OU_MT_MMAPEC_LI, OU_MT_MMAPTC_LI, OU_MTRA_MLCR_LI, OU_MTRA_MLCNR_LI, OU_MTRA_MLCA_LI, OU_MTRA_MLCNA_LI, OU_MTRA_MLCM_LI, OU_MTRA_MLCNM_LI, OU_MTRA_ASPEC_LI, OU_MTRA_ASPTC_LI, OU_MTRA_MMAPEC_LI, OU_MTRA_MMAPTC_LI, <Specific Approach>= External Composition Reuse

Sub-Hypothesis	Measures Used
SH 2.7	OU_MT_SMIN_MLCAEU_LI, OU_MT_SMAX_MLCAEU_LI, OU_MT_SMED_MLCAEU_LI, OU_MT_SMEA_MLCAEU_LI, OU_MT_SMOD_MLCAEU_LI, OU_MT_SMIN_MLCAEIGU_LI, OU_MT_SMAX_MLCAEIGU_LI, OU_MT_SMED_MLCAEIGU_LI, OU_MT_SMEA_MLCAEIGU_LI, OU_MT_SMOD_MLCAEIGU_LI, OU_MTMLCR_LI, OU_MT_MLCNR_LI, OU_MT_MLCA_LI, OU_MT_MLCNA_LI, OU_MT_MLCM_LI, OU_MT_MLCNM_LI, OU_MT_ASPEC_LI, OU_MT_ASPTC_LI, OU_MT_MMAPEC_LI, OU_MT_MMAPTC_LI, OU_MTRA_MLCR_LI, OU_MTRA_MLCNR_LI, OU_MTRA_MLCA_LI, OU_MTRA_MLCNA_LI, OU_MTRA_MLCM_LI, OU_MTRA_MLCNM_LI, OU_MTRA_ASPEC_LI, OU_MTRA_ASPTC_LI, OU_MTRA_MMAPEC_LI, OU_MTRA_MMAPTC_LI, <Specific Approach>= Internal Generation Reuse
SH 2.8	OU_MT_SMIN_MLCAEU_LI, OU_MT_SMAX_MLCAEU_LI, OU_MT_SMED_MLCAEU_LI, OU_MT_SMEA_MLCAEU_LI, OU_MT_SMOD_MLCAEU_LI, OU_MT_SMIN_MLCAEEGU_LI, OU_MT_SMAX_MLCAEEGU_LI, OU_MT_SMED_MLCAEEGU_LI, OU_MT_SMEA_MLCAEEGU_LI, OU_MT_SMOD_MLCAEEGU_LI, OU_MT_MLCR_LI, OU_MT_MLCNR_LI, OU_MT_MLCA_LI, OU_MT_MLCNA_LI, OU_MT_MLCM_LI, OU_MT_MLCNM_LI, OU_MT_ASPEC_LI, OU_MT_ASPTC_LI, OU_MT_MMAPEC_LI, OU_MT_MMAPTC_LI, OU_MTRA_MLCR_LI, OU_MTRA_MLCNR_LI, OU_MTRA_MLCA_LI, OU_MTRA_MLCNA_LI, OU_MTRA_MLCM_LI, OU_MTRA_MLCNM_LI, OU_MTRA_ASPEC_LI, OU_MTRA_ASPTC_LI, OU_MTRA_MMAPEC_LI, OU_MTRA_MMAPTC_LI, <Specific Approach>= External Generation Reuse
SH 2.9	OU_MT_MLCAEU_LI, OU_MT_MLCAEICU_LI, OU_MT_MLCR_LI, OU_MT_MLCNR_LI, OU_MT_MLCA_LI, OU_MT_MLCNA_LI, OU_MT_MLCM_LI, OU_MT_MLCNM_LI, OU_MT_ASPEC_LI, OU_MT_ASPTC_LI, OU_MT_MMAPEC_LI, OU_MT_MMAPTC_LI, OU_MTRA_MLCR_LI, OU_MTRA_MLCNR_LI, OU_MTRA_MLCA_LI, OU_MTRA_MLCNA_LI, OU_MTRA_MLCM_LI, OU_MTRA_MLCNM_LI, OU_MTRA_ASPEC_LI, OU_MTRA_ASPTC_LI, OU_MTRA_MMAPEC_LI, OU_MTRA_MMAPTC_LI, <Specific Approach>= Internal Composition Reuse
SH 2.10	OU_MT_MLCAEU_LI, OU_MT_MLCAEECU_LI, OU_MT_MLCR_LI, OU_MT_MLCNR_LI, OU_MT_MLCA_LI, OU_MT_MLCNA_LI, OU_MT_MLCM_LI, OU_MT_MLCNM_LI, OU_MT_ASPEC_LI, OU_MT_ASPTC_LI, OU_MT_MMAPEC_LI, OU_MT_MMAPTC_LI, OU_MTRA_MLCR_LI, OU_MTRA_MLCNR_LI, OU_MTRA_MLCA_LI, OU_MTRA_MLCNA_LI, OU_MTRA_MLCM_LI, OU_MTRA_MLCNM_LI, OU_MTRA_ASPEC_LI, OU_MTRA_ASPTC_LI, OU_MTRA_MMAPEC_LI, OU_MTRA_MMAPTC_LI, <Specific Approach>= External Composition Reuse
SH 2.11	OU_MT_MLCAEU_LI, OU_MT_MLCAEIGU_LI, OU_MT_MLCR_LI, OU_MT_MLCNR_LI, OU_MT_MLCA_LI, OU_MT_MLCNA_LI, OU_MT_MLCM_LI, OU_MT_MLCNM_LI, OU_MT_ASPEC_LI, OU_MT_ASPTC_LI, OU_MT_MMAPEC_LI, OU_MT_MMAPTC_LI, OU_MTRA_MLCR_LI, OU_MTRA_MLCNR_LI, OU_MTRA_MLCA_LI, OU_MTRA_MLCNA_LI, OU_MTRA_MLCM_LI, OU_MTRA_MLCNM_LI, OU_MTRA_ASPEC_LI, OU_MTRA_ASPTC_LI, OU_MTRA_MMAPEC_LI, OU_MTRA_MMAPTC_LI, <Specific Approach>= Internal Generation Reuse
SH 2.12	OU_MT_MLCAEU_LI, OU_MT_MLCAEEGU_LI, OU_MT_MLCR_LI, OU_MT_MLCNR_LI, OU_MT_MLCA_LI, OU_MT_MLCNA_LI, OU_MT_MLCM_LI, OU_MT_MLCNM_LI, OU_MT_ASPEC_LI, OU_MT_ASPTC_LI, OU_MT_MMAPEC_LI, OU_MT_MMAPTC_LI, OU_MTRA_MLCR_LI, OU_MTRA_MLCNR_LI, OU_MTRA_MLCA_LI, OU_MTRA_MLCNA_LI, OU_MTRA_MLCM_LI, OU_MTRA_MLCNM_LI, OU_MTRA_ASPEC_LI, OU_MTRA_ASPTC_LI, OU_MTRA_MMAPEC_LI, OU_MTRA_MMAPTC_LI, <Specific Approach>= External Generation Reuse

How values for the hypothesis measures support H2 is detailed in the guide to interpretation (Appendix C5). Below are some examples from the guide to interpretation that illustrate this.

To *support* **H2**:

- The highest number of conditional tests in any function, class method, struct method, or template method must be more than zero (MAXCOT_LI > 0).
- The highest number of external calls in any function, class method, struct method, or template method must be more than zero (MAXEC_LI > 0).
- Some specific MLCs mapped to an Automation Element of a given type are used in one or more methodology phases (MLCAEU_LI < 1).
- Some test cases resulted in addition of MLCs (T_MLCA_LI > 0).
- Some test cases resulted in modification of the automation specification prior to execution of any given test case (T_ASPEC_LI > 0).

H2 is *not supported* if:

- The highest number of conditional tests in any function, class method, struct method, or template method equals zero (MAXCOT_LI = 0).
- The highest number of external calls in any function, class method, struct method, or template method equals zero (MAXEC_LI = 0).
- All specific MLCs mapped to an Automation Element of a given type are used in all methodology phase (MLCAEU_LI = 1).
- No test cases resulted in addition of MLCs (T_MLCA_LI = 0).
- No test cases resulted in modification of the automation specification prior to execution of any given test case (T_ASPEC_LI = 0).

The truth of **H2** is *supported* if the truth of **SH 2.1**, **SH 2.2**, and **SH 2.5** are *supported.*

To *support* **SH 2.1**:

- Some specific MLC mapped to Automation Elements are used in some test cases in the software model type classification methodology phase with some software model types and implementations (SMTC_SMEA_MLCAEU_LI < 1).
- Some test cases in the software model type classification methodology phase resulted in removal of MLCs (SMTC_MLCR_LI > 0).

SH 2.1 is *not supported* if:

- All specific MLCs mapped to Automation Elements were used in all test cases in the software model type classification methodology phase with any given software model type and implementation (SMTC_SMEA_MLCAEU_LI = 1).
- No test cases in the software model type classification methodology phase resulted in removal of MLCs (SMTC_MLCR_LI = 0).

To *support* **SH 2.2**:

- Some specific MLC mapped to an Automation Element of a given type are used in one or test cases in the software model classification methodology phase (SMC_MLCAEU_LI < 1).
- Some test cases in the software model classification methodology phase resulted in modification of the automation specification prior to execution of any given test case (SMC_ASPEC_LI > 0).

SH 2.2 is *not supported* if:

- All specific MLCs mapped to an Automation Element of a given type are used in all test case in the software model classification methodology phase (SMC_MLCAEU_LI = 1).
- No test cases in the software model classification methodology phase resulted in modification of the automation specification prior to execution of any given test case (SMC_ASPEC_LI = 0).

To *support* **SH 2.5**:

- Some specific MLCs mapped to Automation Elements are used in some test cases in the measurement testing methodology phase with some given software model types and implementations (MT_SMAX_MLCAEU_LI < 1).
- Some test cases for internal composition reuse in the measurement testing methodology phase resulted in modification of the static part of the meta-model architecture after execution of any given test case (MTRA_MMAPTC_LI > 0).

SH 2.5 is *not supported* if:

- All specific MLCs mapped to Automation Elements are used in all test cases in the measurement testing methodology phase with any given software model type and implementation (MT_SMAX_MLCAEU_LI = 1).
- No test cases for internal composition reuse in the measurement testing methodology phase resulted in modification of the static part of the meta-model architecture after execution of any given test case (MTRA_MMAPTC_LI = 0).

To summarise, **H0b** is *supported* if:

- there are no conditional statements or external calls in the source code, and
- all MLCs are used across all methodology phases with different software model types, and
- no MLCs are added, modified or removed in any methodology phase with different software model types, and
- there are no changes to the source code of the automation specification in any methodology phase with different software model types, and
- there are no changes to the static part of the meta-model architecture in any methodology phase with different software model types, and

H0b is *denied* if:

- there are some conditional statements or external calls in the source code, or
- some MLCs are not used in some methodology phases with different software model types, or
- some MLCs are added, modified, or removed in some methodology phases with different software model types, or
- there are some changes to the source code of the automation specification in some methodology phases with different software model types, or
- there are some changes to the static part of the meta-model architecture in some methodology phases with different software model types, or

H2 is *supported* if:

- There are some conditional statements or external calls in the source code.
- Some MLCs are not used in some methodology phases with different software model types.
- Some MLCs are not used in specific methodology phases, with specific software model types, or specific approaches to reuse.
- Some MLCs are added, modified, or removed in some methodology phases with different software model types.
- Some MLCs are added, modified, or removed in specific methodology phases, with specific software model types, or specific approaches to reuse.
- There are some changes to the source code of the automation specification in some methodology phases with different software model types.
- There are some changes to the source code of the automation specification in specific methodology phases, with specific software model types, or specific approaches to reuse.
- There are some changes to the static part of the meta-model architecture in some methodology phases with different software model types.
- There are some changes to the static part of the meta-model architecture in specific methodology phases, with specific software model types, or specific approaches to reuse.

H2 is *not supported* if:

- There are no conditional statements or external calls in the source code.
- All MLCs are used across all methodology phases with different software model types.
- All MLCs are used in specific methodology phases, with specific software model types, or specific approaches to reuse.
- No MLCs are added, modified, or removed in any methodology phase with different software model types.
- No MLCs are added, modified, or removed in specific methodology phases, with specific software model types, or specific approaches to reuse.
- There are no changes to the source code of the automation specification in any methodology phase with different software model types.

- There are no changes to the source code of the automation specification in specific methodology phases, with specific software model types, or specific approaches to reuse.
- There are no changes to the static part of the meta-model architecture in any methodology phase with different software model types.
- There are no changes to the static part of the meta-model architecture in specific methodology phases, with specific software model types, or specific approaches to reuse.

This concludes the assessment criteria in the automation assessment methodology phase for hypothesis **H0b**, and **H2/SH 2.1 - 2.12**

Analysis Report Input

The methodology phase analysis reports used in this phase are:

The Automation Assessment Analysis Report. See the section under the heading "Automation Assessment Methodology Phase Analysis Report" in Appendix D4 for a complete example.

The hypothesis analysis reports used in this phase are:

The H0 Analysis Report. See the section under the heading "H0 Hypothesis Analysis Report" in Appendix D5 for a complete example.

The H2 Analysis Report. See the section under the heading "H3 Hypothesis Analysis Report" in Appendices D8 – D9 for a complete example.

Analysis Report Output

The analysis reports generated by this methodology phase are:

The Automation Assessment Analysis Report. This report is generated by adding all measures for independent and dependent variables in the automation assessment methodology phase along with the values obtained. In addition, values for all hypothesis measures are added to the report.

The analysis reports modified in this methodology phase are:

The H0 Analysis Report.

The H2 Analysis Report.

The above hypothesis analysis reports include the values obtained for all hypothesis measures from the software model classification methodology phase. Identification of the level of support for each hypothesis is also included and relevant totals for sub-hypotheses and the hypothesis are updated.

Analysis Process Description

1. Calculate the values of the measures derived from the independent variables and dependent variables for the automation assessment methodology phase.
2. Enter the names and values for these measures in the Automation Assessment Methodology Phase Analysis Report.
3. Calculate the values of the base indicators and limitation indicators for hypothesis measures listed under hypothesis measures for the automation assessment methodology phase.
4. Enter the values of the hypothesis measures in the appropriate rows of the Automation Assessment Methodology Phase Analysis report.
5. Copy the values for the hypothesis measures in the Automation Assessment methodology phase analysis report into the appropriate rows in the hypothesis analysis report. For each of theses rows enter the MP_ID in the actual value column as "AA" and add the required values to support and deny the hypothesis.
6. Update the sub-totals and totals in the **H0** hypothesis analysis report, the **H2** hypothesis analysis report, and the Automation Assessment Analysis Report.

4.8 Summary

To summarise:

- The measurement framework claims to measure the amount of reuse for different kinds of software models without any need to change the measurement framework. Four specific claims were made in section 4.3 that amounted to this.
- Four experiments were defined to verify the four claims in section 4.3, there are referred to as methodology phases.
- The experiments were also designed to test the hypotheses using the measurement framework. This is done by using a template for each experiment (methodology phase description template). The template requires each methodology phase to list the hypotheses tested and the meta-level components used from the measurement framework in the experiment.
- Verification that the measurement framework was used to test the hypotheses is done by defining hypotheses measures for each methodology phase, and defining how values for these measures indicate support for the hypotheses. This is detailed in a guide to interpretation for each methodology phase (Appendix C).

This concludes the description of the assessment framework. The next chapter analyses the results obtained after execution of the experiments (methodology phases). The results are contained in the analysis reports are used to evaluate the measurement framework against the claims made about in section 4.3, and to assess the level of support for the hypotheses.

Chapter 5 Data Analysis and Results

This chapter discusses results of the experiments described in chapter 4 (the methodology phases). Results are systematically summarised in analysis reports in Appendix D. A guide to interpretation of hypothesis measures cited in the analysis reports is given in Appendices C2 – C5. Experiment summaries for each experiment are given in Appendix D11.

5.1 Chapter Overview

This chapter is divided into two main sections. These are:

5.2 ***Analysis by Methodology*** phase presents results for each methodology phase based on the claims made in section 4.3 and the hypotheses tested in each methodology phase. Findings can be summarised for each methodology phase as follows:

> ***Software Model Type Classification.*** Results indicate that the measurement framework can classify a range of software model types, including object-oriented and UML software model types.
>
> ***Software Model Classification.*** Results indicate that the measurement framework can use the classifications from the software model type classification methodology phase to classify different software models and measure their size, including object-oriented and UML software model types.
>
> ***Measurement Testing.*** Results indicate that the measurement framework use the classifications from the software model type classification methodology phase to measure the amount of reuse for different software models, including object-oriented and UML software models. However, this is only reliable for analysis and design composition reuse and generation reuse. Modification of the measurement framework may be required for reliable measurement of implementation composition reuse.
>
> ***Automation Assessment.*** Results indicate that the framework has a few limitations. In particular, a different software model type classification is needed for each software model type. A subset of meta-level components were used to support the software model type classification methodology phase. Another subset of meta-level components were only used in the software model classification and measurement testing methodology phases. This suggests some degree of specialised components for the experiments. The number of conditional tests were substantially lower than the number of software model types tested, and the static part of the meta-model architecture and the prototype tool remained the same for each of the methodology phases. This suggests that the framework is general. However, to address the failure cited in the measurement testing methodology phase, modification of the static part of the meta-model architecture and the prototype tool may be necessary.

5.3 ***Analysis by Hypothesis*** summarises results for each hypothesis based on results for each methodology phase to demonstrate the level of support for each

hypothesis. Findings indicate that H0 is not supported, whereas H1 and H2 are supported.

Now follows a more detailed examination of results for each methodology phase.

5.2 Analysis by Methodology Phase

Results presented here are from the methodology phase analysis reports in Appendix D. The guide to interpretation for hypothesis measures in the methodology phase analysis reports is contained in Appendices C2 to C5.

Software Model Type Classification: SMTC

For this experiment, all results for dependent variables, independent variables, and hypothesis measures are contained in the Software Model Type Classification Methodology Phase Analysis Report in Appendix D1. A guide to interpretation for the measures is contained in Appendix C2.

Experiment Results and Claim 1

It was possible to use the measurement framework to classify a range of software model types. However, a number of software model types remain unclassified and represent uncharted territory. A summary of the experiment can be found in Appendix D11.

Results for Hypotheses and Outcomes 1, 2, and 3

Table 5-1 summarises the results of hypothesis measures from the software model type classification methodology phase for the null hypotheses H0a/b.

Table 5-1: Results for Null Hypotheses tested in Software Model Type Classification Phase

Hypothesis Part	Number of Indicators	Number That Support	Percentage That Support	Number that Deny	Percentage That Deny
H0a All SMT	6	0	0%	6	100%
H0a OO/UML SMT	6	0	0%	6	100%
H0b All SMT	10	4	40%	6	60%
H0b OO/UML SMT	10	7	70%	3	30%

The results indicate that H0a/b are not supported. Specifically:

- At least one kind of software model type was classified, and at least one kind of object-oriented or UML software model type was classified (H0a),
- There are limitations with classifying different software model types, and there are limitations with classifying different object-oriented or UML software model types (H0b).

Comparison of summary results for all software model types and OO/UML software model types supports the conclusion that classification of different kinds of software model types appears to have more limitations than classification of object-oriented software model types. However, results for all software model types would be the same as OO/UML software model types if one software model type linked to an implementation were removed from the data set (Structure Model implemented on the XPER C++ tool, SMTI ID: 24).

Table 5-2 summarises the results of hypothesis measures from the software model type classification methodology phase for the hypotheses H1 and H2.

Table 5-2: Results for Sub-Hypotheses tested in Software Model Type Classification Phase

Hypothesis /Sub-Hypothesis	Number of Indicators	Number That Support	Percentage That Support	Number That Don't Support	Percentage That Don't Support
H1/SH 1.1 All SMT	13	13	100%	0	0%
H1/SH 1.1 OO/UML SMT	13	13	100%	0	0%
H2/SH 2.1 All SMT	10	7	70%	3	30%
H2/SH 2.1 OO/UML SMT	10	3	30%	7	70%

Results indicate support for hypothesis H1 (**outcome 1** and **outcome 2**).[1] Successful classification for different software model types as well as different object-oriented and UML software model types was very high Specifically:

- *It was possible to classify different software model types.*

Out of 39 classification attempts for different software model types, all 39 were successful, 100% of classification attempts were successful and 0% were unsuccessful.

- *It was possible to classify different object-oriented and UML software model types.*

Out of 26 classification attempts for different object-oriented or UML software model types, 26 were successful and there were no failures, 100% were successful and 0% were unsuccessful.

[1] (SMTC_BI = 39, SC_CI = 39, UC_CI = 0), (PSC_CI = 100, PUC_CI = 0), (OU_SMTC_BI = 26, OU_SC_CI = 26, OU_UC_CI = 0), (OU_PSC_CI = 100%, OU_PUC_CI = 0%).

Results also indicate support for hypothesis H2 (**outcome 3**).[2] Specifically:

- *Classification of different software model types has limitations.*

Out of 55 software model types only 40, were linked to implementations and used for testing the measurement framework. 71% of software model types were successfully classified, 0% were unsuccessfully classified, the remaining 29% were not classified.

- *Classification of different object-oriented or UML software model types has limitations.*

Out of 35 object-oriented or UML software model types only 26 were linked to implementations and used for testing the measurement framework. 74% of object-oriented or UML software model types were successfully classified, 0% were unsuccessfully classified, the remaining 26% were not classified.

It appears that once a software model type is linked to an implementation it can be classified. The only exception to this was the Structure Model commonly known as the Structure Diagram (Structure Model implemented on the XPER C++ tool, SMTI ID: 24). Although the Structure Model implementation provides some rigour to the software model type, attempts to define an expected classification scheme by hand were unsuccessful and too ambiguous. One difficulty lies in the representation of structure models. The representation is vastly different from most of the software model types that were linked to implementations. Hence, further attempts to classify the structure model were abandoned. The author believes that further research will yield a classification of the structure model. Note that this software model type implementation pair also influenced results for other methodology phases. Details are given under results for each methodology phase.

Comparison of results for all software model types and OO/UML software model types support the interpretation that there were more limitations for classification of different software model types compared to object-oriented or UML software model types. This is due to inclusion of the results for one software model type/implementation pair only, the same pair that created the difference in results for H0 for all software model types and OO/UML software model types (Structure Model implemented on the XPER C++ tool, SMTI ID: 24). This software model type/implementation pair could not be classified. This is indicated by the percentage of successful coverage of software model types with implementations at 98% compared to 100% for object-oriented and UML software model types

[2] (SMT_BI = 55; PCSMTI_LI = 40), (PSCSMT_LI = 71%, PUCSMT_LI = 0%), (OU_SMT_BI = 35; PCSMTI_LI = 26), (OU_PSCSMT_LI = 74%, OU_PUCSMT_LI = 0%).

(PSCSMTI_LI = 98%, OU_PSCSMTI_LI = 100%). If this software model type and implementation is removed then summary results for all software model types would be the same as those for OO/UML software model types. Given that this is the only exception, it is more likely that the limitations discovered for classification of different software model types are the same as those for object-oriented or UML software model types.

More significant is the number of software model types that are untested and remain an open question for further research. These software model types include:

- A number of non-object-oriented software model types (Petri Nets [1-3], Rich Pictures [4, 5], , Business Rule Models [6, 7], the Living Systems Model (LSM) [8, 9] and Software Execution Graphs [10]).
- A number of object-oriented software model types (Timethread-Role Maps [11], Task Models [12], Architecture Models [12], MOOD [13], Multiple Interface Object Model [14] [15, 16], the CO-IP Model [17], the OSA Model [18], Collaborations Models [19], OOSDL [20]).
- Some programming languages (the FORK language [21], the OLI language [22]).

Although the programming language implementations could be obtained, tests conducted during the measurement testing methodology phase with other programming languages illustrated a more fundamental limitation found in the measurement testing methodology phase. To overcome this limitation further research is required beyond this thesis. More testing of languages would only illustrate the same limitation.

Software Model Classification: SMC

For this experiment, all results for dependent variables, independent variables, and hypothesis measures are contained in the Software Model Classification Methodology Phase Analysis Report in Appendix D2. A guide to interpretation for the measures is contained in Appendix C3. Footnotes are used to cite the hypothesis measures and their values from the Software Model Classification Methodology Phase Analysis Report.

Experiment Results and Claim 2

In this experiment, a range of software models were successfully classified and measured for their size. This includes results for object-oriented and UML software model types. However, the software model types that were not classified in the software model type classification methodology phase also have consequences in this methodology phases. We are unable to determine if the measurement framework can classify software models based on these software model types or measure their size. A summary of the experiment can be found in Appendix D11.

Results for Hypotheses and Outcomes 1 to 6

Table 5-3 summarises the results of hypothesis measures from the software model classification methodology phase for the null hypotheses H0a/b.[3]

Table 5-3: Results for Hypothesis Parts tested in Software Model Classification Phase

Hypothesis Part	Number of Indicators	Number That Support	Percentage That Support	Number that Deny	Percentage That Deny
H0a All SMT	4	0	0%	4	100%
H0a OO/UML SMT	4	0	0%	4	100%
H0b All SMT	14	6	43%	8	57%
H0b OO/UML SMT	14	7	50%	7	50%

[3] This page is part of a Thesis submitted in the year 2002 by Eugene Eric Doroshenko in fulfilment of the requirements for a Doctor of Philosophy (Information Systems), at the University of Tasmania. If you are reading this page and it is not part of a Thesis by Eugene Doroshenko then you should contact the Secretary, Board of Graduate Studies by Research, University of Tasmania, Churchill Avenue, Sandy Bay, GPO Box 252-45, Hobart 7001, Tasmania, Australia. Telephone +61-3-6226-2762. Fax +61-3-6226-7497. email secretary.bsgr@utas.edu.au.

Results indicate that H0a/b are not supported. Specifically:

- At least one software model can be successfully classified and measurement of size is also successful for at least one software model type, and at least one object-oriented or UML software model can be successfully classified and measurement of size is also successful for at least one object-oriented or UML software model type (H0a).
- There are some limitations with classifying and measuring the size of different software models and software model types, and there are some limitations with classifying and measuring the size of different object-oriented or UML software models and software model types (H0b).

The discrepancy between results for all software model types and OO/UML software model types is due to one software model type linked to an implementation that could not be classified in the software model type classification methodology phase, namely the Structure Model (SMTI ID = 24). The software model type was not categorised as a UML or object-oriented software model type. This discrepancy is manifest in the results for two indicators (CSMTI_LI for all software model types and OU_CSMTI_LI for OO/UML software model types). But for this software model type, results for all software model types and OO/UML software model types would be the same.

Table 5-4 summarises the results of hypothesis measures from the software model classification methodology phase for the hypotheses H1 and H2.

Table 5-4: Results for Sub-Hypotheses tested in Software Model Classification Phase

Hypothesis /Sub-Hypothesis	Number of Indicators	Number That Support	Percentage That Support	Number That Don't Support	Percentage That Don't Support
H1/SH 1.2 All SMT	5	5	100%	0	0%
H1/SH 1.3 All SMT	5	5	100%	0	0%
H1/SH 1.4 All SMT	5	5	100%	0	0%
H1/SH 1.2 OO/UML SMT	5	5	100%	0	0%
H1/SH 1.3 OO/UML SMT	5	5	100%	0	0%
H1/SH 1.4 OO/UML SMT	5	5	100%	0	0%
H2/SH 2.2 All SMT	8	5	38%	3	62%
H2/SH 2.3 All SMT	6	3	50%	3	50%
H2/SH 2.4 All SMT	4	1	25%	3	75%
H2/SH 2.2 OO/UML SMT	8	4	50%	4	50%
H2/SH 2.3 OO/UML SMT	6	3	50%	3	50%
H2/SH 2.4 OO/UML SMT	4	1	25%	3	75%

Results indicate support for H1 (**outcome 1** and **outcome 2**). There were no failures at either classification or measurement of size for software models with successful software model type classifications.[4] Specifically:

- *It is possible to classify and measure the size of different software models based on their type using the measurement framework.*

Out of 39 software model type/implementation pairs tested, 100% of attempts at classification and size measurement were successful. The number of test cases was about 18,000 for all software model types (SMTTCR_BI = 18317).

- *It is possible to classify and measure the size of different object-oriented and UML software models based on their type using the measurement framework.*

[4] (SMTTC_BI = 39, PSMC_CI = 100%, PSMS_CI = 100%), (OU_SMTTC_BI = 26, OU_PSMC_CI = 100%, OU_PSMS_CI = 100%).

Out of 26 object-oriented and UML software model type/implementation pairs tested, 100% of attempts at classification and size measurement were successful. The number of test cases was about 13,000 for object-oriented or UML software model types (OU_SMTTCR_BI = 13866).

It is fair to say that if a software model type can be classified then it is possible to classify software models of the same type and measure their size using the measurement framework.

Results for H2 indicate support for H2 (**outcomes 3 – 6**). There are some limitations identified in the measurement framework when classifying and measuring sizes of different software models and different kinds of software models (H2/SH 2.2 - 2.4), and when classifying and measuring sizes of different object-oriented and UML software models (H2/SH 2.2 - 2.4). Specifically:

- *A number of software model types remain untested with respect to classification and measurement of size*[5] (**outcome 3**).

Out of 55 software model types only 40 were linked to implementations and were available for tests. Out of 35 object-oriented or UML software model types only 26 were linked to implementations and were available for tests.

- *No successful test cases for classification of software models or measurement of size could be found for software model types with unsuccessful software model type classifications*[6] (**outcome 4**).

This was the same for object-oriented and UML software model types. It might be inferred that a successful software model type classification is required for successful classification and measurement of size of software models. However, there were no software model types with unsuccessful classifications available for tests.

- *A significant limitation is the dependency on different software model type classifications for successful classification of software models and measurement of their size*[7] (**outcome 5 & 6**).

If any two software models are based on different software model types, then different software model type classifications are required for successful classification of these software models and measurement of their size. If any two software model

---

[5] (SMT_BI = 55, PCSMT_LI = 40), (OU_SMT_BI = 35, OU_PCSMT_LI = 26).

[6] (SMUCST_LI = 0, SMUCSM_LI = 0), (OU_SMUCST_LI = 0, OU_SMUCSM_LI = 0).

[7] (SMTTCP_BI = 741, SCTCCTC_LI = 1, SCTCDTC_LI = 740), (OU_SMTTCP_BI = 325, OU_SCTCCTC_LI = 1, OU_SCTCDTC_LI = 324).

type classifications are compared, out of 741 pairs of different software model types linked to implementations, only 1 pair used the same software model type classification scheme and the remaining 740 pairs used different ones. Out of 325 pairs of different object-oriented or UML software model types linked to implementations, only 1 pair used the same software model type classification scheme and the remaining 324 pairs used different ones.

With respect to **outcome 4**, the discrepancy between all software model types and OO/UML software model types is due to one software model type linked to an implementation (the Structure Diagram, SMT ID: 24). Out of 40 software model types linked to implementations 39 were classified. Out of 26 object-oriented or UML software model types all 26 were classified.[8] The software model type was named the Structure Model, commonly referred to as the Structure Diagram (SMTI ID = 24). It was not possible to even define an expected classification scheme for this software model type even though it was linked to an implementation. Reasons for this were given under the previous phase (Software model type classification). Removal of this software model type would have made summary results for all software model types and OO/UML software model types identical in Table 5-4.

With respect to **outcome 5 & 6**, the two software model type/implementations that used the same software model type classification scheme were IDL implemented on Paradigm Plus® (SMTI ID = 19) and IDL implemented on the Orbakus Object Request Broker® (SMTI ID = 20). Although the implementations were different the software model type itself was the same. This was done because a different implementation was used to test generation reuse and composition reuse with IDL in the measurement testing methodology phase. Specifically, the Orbakus Object Request Broker was used to test measurement of composition reuse and Paradigm Plus was used to test measurement of generation reuse. Thus, it appears that successful classification software models and measurement of their size for different software model types requires different software model type classification schemes.

[8] (SMTI_BI = 40, CSMTI_LI = 39), (OU_SMTI_BI = 26, OU_CSMTI_LI = 26).

Measurement Testing: MT

For this experiment, all results for dependent variables, independent variables, and hypothesis measures are contained in the Measurement Testing Methodology Phase Analysis Report in Appendix D3. A guide to interpretation for the measures is contained in Appendix C4. Footnotes are used to cite the hypothesis measures and their values from the Measurement Testing Methodology Phase Analysis Report.

Experiment Summary and Claim 3

In this experiment the amount of reuse was successfully measured for a range of software models. However, there were failures even with software model types that were successfully classified. A summary of the experiment can be found in Appendix D11.

Results for Hypotheses and Outcomes 1 to 6

Table 5-5 summarises the results of hypothesis measures from the measurement testing phase for the null hypotheses H0a/b.

Table 5-5: Results for Hypothesis Parts tested in Measurement Testing Phase

Hypothesis Part	Number of Indicators	Number That Support	Percentage That Support	Number that Deny	Percentage That Deny
H0a All SMT	10	0	0%	10	100%
H0a OO/UML SMT	10	0	0%	10	100%
H0b All SMT	8	1	12%	7	88%
H0b OO/UML SMT	8	2	25%	6	75%

Results indicate that hypotheses H0a/b are not supported. Specifically:

- Measurement of the amount of reuse is successful for at least one software model and one software model type, and at least one object-oriented or UML software model and one object-oriented or UML software model type (H0a).
- There are some limitations with measuring the amount of reuse of different software models and software model types, and different object-oriented or UML software models and software model types (H0b).

The discrepancy between results for all software model types and OO/UML software model types is due to one software model type linked to an implementation that could not be classified in the software model type classification methodology phase, namely the Structure Model (SMTI ID = 24). The software model type was not

categorised as a UML or object-oriented software model type. This discrepancy is manifest in the results for two indicators (CSMTI_LI for all software model types and OU_CSMTI_LI for OO/UML software model types). But for this software model type, results for all software model types and OO/UML software model types would be the same.

Table 5-6 and Table 5-7 summarise the results of hypothesis measures from the measurement testing methodology phase for the hypotheses H1.

Table 5-6: Results for Sub-Hypotheses tested for H1 in Measurement Testing Phase (All Software Model Types)

Hypothesis /Sub-Hypothesis	Number of Indicators	Number That Support	Percentage That Support	Number That Don't Support	Percentage That Don't Support
H1/SH 1.5	10	10	100%	0	0%
H1/SH 1.6	10	10	100%	0	0%
H1/SH 1.7	10	10	100%	0	0%
H1/SH 1.8	10	10	100%	0	0%
H1/SH 1.9	10	10	100%	0	0%
H1/SH 1.10	10	10	100%	0	0%
H1/SH 1.11	10	10	100%	0	0%
H1/SH 1.12	10	10	100%	0	0%

Table 5-7: Results for Sub-Hypotheses tested for H1 in Measurement Testing Phase (OO/UML Software Model Types)

Hypothesis /Sub-Hypothesis	Number of Indicators	Number That Support	Percentage That Support	Number That Don't Support	Percentage That Don't Support
H1/SH 1.5	10	10	100%	0	0%
H1/SH 1.6	10	10	100%	0	0%
H1/SH 1.7	10	10	100%	0	0%
H1/SH 1.8	10	10	100%	0	0%
H1/SH 1.9	10	10	100%	0	0%
H1/SH 1.10	10	10	100%	0	0%
H1/SH 1.11	10	10	100%	0	0%
H1/SH 1.12	10	10	100%	0	0%

Results indicate support for H1 (**outcome 1 and 2**). There were failures for a selection of software model types but the degree of success remains high.[9] Specifically:

- *It is possible to measure internal composition reuse, external composition reuse, internal generation reuse, and external generation reuse for different software models and different software model types.*

39 software model type/implementation pairs were used for testing measurement of internal composition reuse, external composition reuse, internal generation reuse, and external generation reuse. This resulted in 84 test data sets. 92 % of test cases were successful and only 8% were unsuccessful for all software model types.

- *It is possible to measure internal composition reuse, external composition reuse, internal generation reuse, and external generation reuse for different object-oriented and UML software models and different object-oriented and UML software model types*

26 object-oriented or UML software model type/implementation pairs were used for testing measurement of internal composition reuse, external composition reuse, internal generation reuse, and external generation reuse. This resulted in 56 test data sets. 94 % of test cases were successful and only 6% were unsuccessful for all software model types.

[9] (SMTC_BI = 39, MTTC_BI = 84), (PSRM_CI = 92%, PURM_CI = 8%), (OU_SMTC_BI = 26, OU_MTTC_BI = 56), (OU_PSRM_CI = 92%, OU_PURM_CI = 8%).

It is fair to say that if a software model type can be classified then it is possible to measure the amount of reuse with a high degree of success. Examination of indicators for internal composition reuse and external composition reuse further support this.[10] Specifically:

- 38 software model type/implementation pairs were used for testing of internal composition reuse and external composition reuse. 25 software model type/implementation pairs used were object-oriented or UML software model types.
- 91% of test cases were successful and only 9% were unsuccessful for all software model types.
- A similar result was achieved for object-oriented and UML software model types. 93% of test cases were successful and only 7% were unsuccessful.
- There were about 5,000 test cases for internal composition reuse and 5,000 test cases for external composition reuse using all software model types
- There were about 3,000 test cases internal composition reuse and about 3,000 test cases for external composition reuse using object-oriented and UML software model types.

Examination of indicators for internal generation reuse and external generation reuse does reveal some slightly differing findings.

[10] (MTTCIC_BI = 38, MTTCEC_BI = 38), (OU_MTTCIC_BI = 25, OU_MTTCEC_BI = 25), (ICPSRM_CI = 91%, ECPSRM_CI = 91%, ICPURM_CI = 9%, ECPURM_CI = 9%), (OU_ICPSRM_CI = 93%, OU_ECPSRM_CI = 93%, OU_ICPURM_CI = 7%, OU_ECPURM_CI = 7%), (MTTCICR_BI = 5624, MTTCECR_BI = 5624, OU_MTTCICR_BI = 3700, OU_MTTCECR_BI = 3700).

The degree of success for measurement of generation reuse was high but the number of test data sets was much lower compared to composition reuse.[11] Specifically:

- Only 4 software model type/implementation pairs were used for testing of internal generation reuse and external generation reuse. Of these 3 software model type/implementation pairs used were object-oriented or UML software model types.
- 100% of test cases were successful for all software model types. The same result was achieved for object-oriented and UML software model types. 100% of test cases were successful.
- There were about 500 test cases for internal generation reuse and about 500 test cases for external generation reuse using all software model types.
- There were about 400 test cases for internal generation reuse and about 500 test cases for external generation reuse using object-oriented and UML software model types.

It may be fair to say that if a software model type can be classified then it is possible to measure internal and external generation reuse using the measurement framework. However, it may also be necessary to gather more test data sets to really support this claim.

There are two reasons why so few test data sets were obtained. Firstly, it takes a long time to classify implementation models that are generated from analysis or design models to get 100% success. Secondly, out of the implementation models chosen, only a few had implementations (CASE tools) that generated them.

A closer examination of the source of failures supports the following conclusions.

- *If a software model type that is classified is an analysis or design model then it is possible to measure the amount of internal and external composition reuse using the measurement framework.*

If the results for measurement of internal and external composition measurement for software model type/implementation pairs that are programming languages or text based are excluded then the results are the same as those for generation reuse. That

---

[11] (MTTCIG_BI = 4, MTTCEC_BI = 4), (OU_MTTCIC_BI = 3, OU_MTTCEC_BI = 3), (IGPSRM_CI = 91%, EGPSRM_CI = 100%, IGPURM_CI = 0%, EGPURM_CI = 0%), (OU_IGPSRM_CI = 100%, OU_EGPSRM_CI = 100%, OU_IGPURM_CI = 0%, OU_EGPURM_CI = 0%), (MTTCICR_BI = 592, MTTCECR_BI = 592, OU_MTTCICR_BI = 444, OU_MTTCECR_BI = 444).

is, 100% success for measurement of the amount of reuse using internal composition, external composition, internal generation, and external generation (the selection of software model type/implementation pairs that are programming languages or text based have the following SMTI ID values: 17, 18, 20, 25, 26, 34).

- *If a software model that is classified is an implementation model then it is not possible to measure the amount of internal and external composition reuse using the measurement framework with any high degree of success.*

If the same selection of software model type/implementation pairs (SMTI ID values: 17, 18, 20, 25, 26, 34) is analysed separately these results are not positive. For internal and external composition reuse only 43% of test cases were successful with 57% of test cases unsuccessful for the selection of software model types as well as object-oriented and UML software model types contained in the selection.

Table 5-8: Results for Sub-Hypotheses tested for H2 in Measurement Testing Phase (All Software Model Types)

Hypothesis /Sub-Hypothesis	Number of Indicators	Number That Support	Percentage That Support	Number That Don't Support	Percentage That Don't Support
H2/SH 2.5	14	12	86%	2	14%
H2/SH 2.6	14	12	86%	2	14%
H2/SH 2.7	14	10	71%	4	29%
H2/SH 2.8	14	10	71%	4	29%
H2/SH 2.9	8	6	75%	2	25%
H2/SH 2.10	8	6	75%	2	25%
H2/SH 2.11	8	4	50%	4	50%
H2/SH 2.12	8	4	50%	4	50%

Table 5-9: Results for Sub-Hypotheses tested for H2 in Measurement Testing Phase (OO/UML Software Model Types)

Hypothesis /Sub-Hypothesis	Number of Indicators	Number That Support	Percentage That Support	Number That Don't Support	Percentage That Don't Support
H2/SH 2.5	14	11	79%	3	21%
H2/SH 2.6	14	11	79%	3	21%
H2/SH 2.7	14	9	64%	5	36%
H2/SH 2.8	14	9	64%	5	36%
H2/SH 2.9	8	6	75%	2	25%
H2/SH 2.10	8	6	75%	2	25%
H2/SH 2.11	8	4	50%	4	50%
H2/SH 2.12	8	4	50%	4	50%

This contrast in results may be due to representation of the models. Implementation models are usually represented as text whereas analysis and design models are usually represented using graphics. However, the author believes that more accurate results for implementation models will require modification of the static part meta-model architecture. Actual results for implementation models were only accurate if the expected results for amount reused or the amount not reused were equal to zero.

Table 5-8 and Table 5-9 summarise results for the hypothesis measures from the measurement testing methodology phase for the hypothesis H2.

Results indicate support for H2 (**outcomes 3 – 6**). There were some limitations identified in the measurement framework when measuring the amount of internal composition reuse, external composition reuse, internal generation reuse, and external generation reuse with different software models and software model types and different object-oriented and UML software models and software model types (H2/SH 2.5 - 2.12). Specifically

- *A number of software model types remain untested with respect to measurement of reuse*[12] (**outcome 3**).

For both software model types in general and object-oriented or UML software model types a number of software model types were not classified because they were not linked to implementations. Out of 55 software model types only 40 were linked to implementations and were available for tests. Out of 35 object-oriented or UML software model types only 26 were linked to implementations and were available for tests.

- *Successful software model type classification does not guarantee successful measurement of reuse but is still dependent on it*[13] (**outcome 4**).

No successful test cases for measurement of reuse could be found for software model types with unsuccessful software model type classifications. This was the same for object-oriented and UML software model types. It might be inferred that a successful software model type classification is required for successful measurement of reuse of software models. However, there were no software model types with unsuccessful classifications available for tests. More significant is the number of unsuccessful test cases for software models with successful software model type classifications. About 1000 test cases were unsuccessful for software model type/implementation pairs with successful software model type classifications. About 500 test cases were

---

[12] (SMT_BI = 55, PCSMT_LI = 40), (OU_SMT_BI = 35, OU_PCSMT_LI = 26).

[13] (UCSRM_LI = 0), (OU_UCSRM_LI = 0), (SCURM_LI = 1020), (OU_SCURM_LI = 510).

unsuccessful for object-oriented and UML software models with successful software model type classifications.

- *Successful software model type classification does not guarantee successful measurement of composition reuse*[14] (**outcome 4**).

Indicators specific to composition reuse support this. The number of unsuccessful test cases for software models with successful software model type classifications was 510 for internal and external composition reuse. For object-oriented and UML software models the number of unsuccessful test cases was 255 for internal and external composition reuse.

- *Successful software model type classification appears to guarantee successful measurement of generation reuse*[15] (**outcome 4**).

Indicators specific to generation reuse support this. The number of unsuccessful test cases for software models with successful software model type classifications was 0 for internal and external generation reuse. For object-oriented and UML software models the number of unsuccessful test cases was 0 for internal and external generation reuse.

- *If any two software models are based on a different software model type, then different software model type classifications are required for successful measurement of reuse for their respective software models*[16] (**outcome 5 & 6**).

A significant limitation is the dependency on different software model type classifications for successful measurement of reuse. For any two software model type classifications used in measurement testing for a given approach to reuse, only 1 out of 1418 pairs used the same software model type classification scheme and the remaining 1417 pairs used different ones. Out of 606 pairs of different object-oriented or UML software model type classifications used in measurement testing for a given approach to reuse, only 1 out of 606 pairs used the same software model type classification scheme and the remaining 605 pairs used different ones.

---

[14] (ICSCURM_LI = 510, ECSCURM_LI = 510), (OU_ICSCURM_LI = 255, OU_ECSCURM_LI = 255).

[15] (IGSCURM_LI = 0, EGSCURM_LI = 0), (OU_IGSCURM_LI = 0, OU_EGSCURM_LI = 0).

[16] (MTTCP_BI = 1418, SRTCCTC_LI = 1, SRTCDTC_LI = 1417), (OU_MTTCP_BI = 606, OU_SRTCCTC_LI = 1, OU_SRTCDTC_LI = 605).

- *Indicators specific to the reuse approach further support the view that different software model type classifications are required for successful measurement of reuse for their respective software models*[17] **(outcome 5 & 6)**.

Out of 703 pairs of different software model types linked to implementations for internal and external composition reuse measurement, all 703 pairs used different software model type classifications. Out of 300 pairs of different object-oriented or UML software model types linked to implementations, all 300 pairs used different software model type classifications. Out of 6 pairs of different software model types linked to implementations for internal and external generation reuse measurement, all 6 pairs used different software model type classifications. Out of 3 pairs of different object-oriented or UML software model types linked to implementations, all 3 pairs used different software model type classifications.

With respect to **outcome 4**, the discrepancy between results for all software model types and OO/UML software model types is due to one software model type linked to an implementation that could not be classified in the software model type classification methodology phase, namely the Structure Model (SMTI ID = 24). This discrepancy is manifest in the results for two indicators (CSMTI_LI for H3 and OU_CSMTI_LI for H4). But for this software model type, results for all software model types and OO/UML software model types would be the same.

With respect to **outcomes 5 & 6**, the two software model type/implementations that used the same software model type classification scheme were IDL implemented on Paradigm Plus® (SMTI ID = 19) and IDL implemented on the Orbakus Object Request Broker® (SMTI ID = 20). Although the implementations were different the software model type itself was the same. This was done because a different implementation was used to test generation reuse and composition reuse with IDL in the measurement testing methodology phase. Thus, it appears that successful measurement of reuse for software model types requires different software model type classification schemes.

---

[17] (MTTCICP_BI = 703, ICSRTCCTC_LI = 0, ICSRTCDTC_LI = 703, MTTCECP_BI = 703, ECSRTCCTC_LI = 0, ECSRTCDTC_LI = 703), (OU_MTTCICP_BI = 300, OU_ICSRTCCTC_LI = 0, OU_ICSRTCDTC_LI = 300, OU_MTTCECP_BI = 300, OU_ECSRTCCTC_LI = 0, OU_ECSRTCDTC_LI = 300), (MTTCIGP_BI = 6, IGSRTCCTC_LI = 0, IGSRTCDTC_LI = 6, MTTCEGP_BI = 6, EGSRTCCTC_LI = 0, EGSRTCDTC_LI = 6), (OU_MTTCIGP_BI = 3, OU_IGSRTCCTC_LI = 0, OU_IGSRTCDTC_LI = 3, OU_MTTCEGP_BI = 3, OU_EGSRTCCTC_LI = 0, OU_EGSRTCDTC_LI = 3).

Automation Assessment: AA

For this experiment, all results for dependent variables, independent variables, and hypothesis measures are contained in the Automation Assessment Methodology Phase Analysis Report in Appendix D4. A guide to interpretation for the measures is contained in Appendix C5.

Experiment Summary and Claim 4

Results indicate that the measurement framework was able to measure the amount of reuse for a range of software model types without any changes to the static part of the measurement framework. In addition, the same meta-level components were used to do this. A summary of this experiment can be found in Appendix D11.

Results for Hypotheses and Outcomes 1 to 5

Table 5-10 summarises the results of hypothesis measures from the automation assessment methodology phase for the null hypothesis H0b.

Table 5-10: Results for Hypothesis Parts tested in Automation Assessment Phase

Hypothesis Part	Number of Indicators	Number That Support	Percentage That Support	Number that Deny	Percentage That Deny
H0b All SMT	68	32	47%	36	53%
H0b OO/UML SMT	58	24	41%	34	59%

Results indicate that hypothesis H0b is not supported. Specifically:

- There are limitations for measurement of the amount of reuse with different software model types, including object-oriented and UML software model types, using an automated version of the measurement framework (the prototype tool).

The discrepancy between all software model types and OO/UML software model types is due largely to a set of ten hypothesis measures that can only be applied to all software model types and cannot be applied to a subset of software model types used[18] (for instance, object-oriented and UML software model types). However, all object-oriented and UML software model types used were included for calculation of these hypothesis measures.

[18] MINCOT_LI, MAXCOT_LI, MEACOT_LI, MEDCOT_LI, MODCOT_LI, MINEC_LI, MAXEC_LI, MEAEC_LI, MEDEC_LI, and MODEC_LI.

Table 5-11 and Table 5-12 summarise the results of hypothesis measures from the automation assessment methodology phase for hypothesis H2.

Table 5-11: Results for Sub-Hypotheses tested for H2 in Automation Assessment Phase (All Software Model Types)

Sub-Hypothesis	Number of Indicators	Number That Support	Percentage That Support	Number That Don't Support	Percentage That Don't Support
SH 2.1 - 2.12	68	35	51%	33	49%
SH 2.1	50	29	58%	21	42%
SH 2.2	18	1	6%	17	94%
SH 2.3	50	5	10%	45	90%
SH 2.4	18	1	6%	17	94%
SH 2.5	100	10	10%	90	90%
SH 2.6	100	10	10%	90	90%
SH 2.7	100	10	10%	90	90%
SH 2.8	100	10	10%	90	90%
SH 2.9	36	2	6%	34	94%
SH 2.10	36	2	6%	34	94%
SH 2.11	36	2	6%	34	94%
SH 2.12	36	2	6%	34	94%

Table 5-12: Results for Sub-Hypotheses tested for H2 in Automation Assessment Phase (OO/UML Software Model Types)

Sub-Hypothesis	Number of Indicators	Number That Support	Percentage That Support	Number That Don't Support	Percentage That Don't Support
SH 2.1 - 2.12	58	34	59%	24	41%
SH 2.1	50	29	58%	21	42%
SH 2.2	18	1	6%	17	94%
SH 2.3	50	5	10%	45	90%
SH 2.4	18	1	6%	17	94%
SH 2.5	100	10	10%	90	90%
SH 2.6	100	10	10%	90	90%
SH 2.7	100	10	10%	90	90%
SH 2.8	100	10	10%	90	90%
SH 2.9	36	2	6%	34	94%
SH 2.10	36	2	6%	34	94%
SH 2.11	36	2	6%	34	94%
SH 2.12	36	2	6%	34	94%

Results for H2 indicate some support for H2. Note that when H2 is not supported this suggests that the measurement framework is a general framework. Specifically:

- *It is unlikely that a separate set of code in the automation specification is used to classify different software models to measure the amount of reuse for each software model type* [19] **(outcome 1)**.

Relative to the number of software model types used (39), the vast majority of class methods, functions, and template methods have total conditional tests at zero. The mean value for this is 0.49 and the median is 0. None of the class methods, template methods, or functions had any external calls. This eliminated the requirement to examine a different tool to support the need to use different blocks of code for different software model types via another software tool. The maximum number of conditional tests was 9 and the most frequently occurring value was zero.

- *A subset of meta-level components was used in the software model type classification methodology phase*[20] **(outcome 2)**.

More specifically the software model set theory MLC, software model classification MLC, amount of reuse measurement model specifier MLC, and measurement data classification MLC were not used in some phases with different software model types. The results were the same for object-oriented and UML software model types (See Table 5-13). Further examination of results for specific phases sheds more light in the reason for the difference. Hypothesis measures for use of meta-level components in the software model type classification methodology phase show that the software model type classification MLC, software model type set theory MLC, and amount of reuse measurement M1 model specifier MLC were the only meta-level components used for that phase (See Table 5-14). This contrasts sharply with results for the software model classification methodology phase and measurement testing methodology phase (Compare Table 5-15 with Table 5-14). The same results were obtained for object-oriented and UML software model types (Compare Table 5-16 and Table 5-17). In addition four meta-level components were only used to classify different software models and measure their size (software model

---

[19] (SMTIR_BI = 39), (MEACOT_LI = 0.49, MEDCOT_LI = 0), (MINEC_LI = 0, MAXEC_LI = 0, MEDEC_LI = 0, MEAEC_LI = 0, MODEC_LI = 0), (MAXCOT_LI = 9, MODCOT_LI = 0).

[20] That is, results for SMIN_MLCAEU_LI, SMAX_MLCAEU_LI, SMED_MLCAEU_LI, SMEA_MLCAEU_LI, and SMOD_MLCAEU_LI are similar to results for OU_SMIN_MLCAEU_LI, OU_SMAX_MLCAEU_LI, OU_SMED_MLCAEU_LI, OU_SMEA_MLCAEU_LI, and OU_SMOD_MLCAEU_LI.

classification), and measure the amount of reuse (measurement testing). The four meta-level components are the software model set theory MLC, software model classification MLC, amount of reuse measurement model specifier MLC, and the measurement data classification MLC.

Table 5-13: Results for use of meta-level components

MLC Name	All Software Model Types (MLCAEU_LI)	OO/UML Software Model Types (OU_MLCAEU_LI)
Amount of Reuse Measurement M2 Model Specifier	0.000	0.000
Software Model Type Set Theory	1.000	1.000
Software Model Type Classification	1.000	1.000
Amount of Reuse Measurement M1 Model Specifier	1.000	1.000
Software Model Set Theory	0.999	0.999
Software Model Classification	0.999	0.999
Amount of Reuse Measurement Model Specifier	0.999	0.999
Measurement Data Classification	0.999	0.999

Table 5-14: Results summary for usage indicators in software model type classification methodology phase.

MLC Name	Result (SMTC_SMIN_MLCAEU_LI, SMTC_SMAX_MLCAEU_LI, SMTC_SMED_MLCAEU_LI, SMTC_SMEA_MLCAEU_LI, SMTC_SMOD_MLCAEU_LI)
Amount of Reuse Measurement M2 Model Specifier	0.000
Software Model Type Set Theory	1.000
Software Model Type Classification	1.000
Amount of Reuse Measurement M1 Model Specifier	1.000
Software Model Set Theory	0.000
Software Model Classification	0.000
Amount of Reuse Measurement Model Specifier	0.000
Measurement Data Classification	0.000

Table 5-15: Results summary for usage indicators in the software model classification and measurement testing methodology phases.

MLC Name	Result (SMC) (SMC_SMIN_MLCAEU_LI, SMC_SMAX_MLCAEU_LI, SMC_SMED_MLCAEU_LI, SMC_SMEA_MLCAEU_LI, SMC_SMOD_MLCAEU_LI)	Result (MT) (MT_SMIN_MLCAEU_LI, MT_SMAX_MLCAEU_LI, MT_SMED_MLCAEU_LI, MT_SMEA_MLCAEU_LI, MT_SMOD_MLCAEU_LI)
Amount of Reuse Measurement M2 Model Specifier	0.000	0.000
Software Model Type Set Theory	1.000	1.000
Software Model Type Classification	1.000	1.000
Amount of Reuse Measurement M1 Model Specifier	1.000	1.000
Software Model Set Theory	1.000	1.000
Software Model Classification	1.000	1.000
Amount of Reuse Measurement Model Specifier	1.000	1.000
Measurement Data Classification	1.000	1.000

Table 5-16: Results summary for usage indicators in software model type classification methodology phase (object-oriented and UML software model types).

MLC Name	Result (OU_SMTC_SMIN_MLCAEU_LI, OU_SMTC_SMAX_MLCAEU_LI, OU_SMTC_SMED_MLCAEU_LI, OU_SMTC_SMEA_MLCAEU_LI, OU_SMTC_SMOD_MLCAEU_LI)
Amount of Reuse Measurement M2 Model Specifier	0.000
Software Model Type Set Theory	1.000
Software Model Type Classification	1.000
Amount of Reuse Measurement M1 Model Specifier	1.000
Software Model Set Theory	0.000
Software Model Classification	0.000
Amount of Reuse Measurement Model Specifier	0.000
Measurement Data Classification	0.000

Table 5-17: Results summary for usage indicators in the software model classification and measurement testing methodology phase (object-oriented and UML software model types).

MLC Name	Result (SMC) (OU_SMC_SMIN_MLCAEU_LI, OU_SMC_SMAX_MLCAEU_LI, OU_SMC_SMED_MLCAEU_LI, OU_SMC_SMEA_MLCAEU_LI, OU_SMC_SMOD_MLCAEU_LI)	Result (MT) (OU_MT_SMIN_MLCAEU_LI, OU_MT_SMAX_MLCAEU_LI, OU_MT_SMED_MLCAEU_LI, OU_MT_SMEA_MLCAEU_LI, OU_MT_SMOD_MLCAEU_LI)
Amount of Reuse Measurement M2 Model Specifier	0.000	0.000
Software Model Type Set Theory	1.000	1.000
Software Model Type Classification	1.000	1.000
Amount of Reuse Measurement M1 Model Specifier	1.000	1.000
Software Model Set Theory	1.000	1.000
Software Model Classification	1.000	1.000
Amount of Reuse Measurement Model Specifier	1.000	1.000
Measurement Data Classification	1.000	1.000

- *The dynamic part of the meta-model architecture requires some modification to measure the amount of reuse with different kinds of software models, including object-oriented and UML software models*[21] **(outcome 3)**.

A number of test cases included addition or modification of meta-level components for the dynamic part of the meta-model architecture for different software model types. Similar results were found for different object-oriented and UML software model types.

[21] (T_MLCA_LI = 39, T_MLCMOD_LI = 19), (OU_T_MLCA_LI = 26, OU_T_MLCMOD_LI = 14).

Examination of indicators for specific methodology phases yields a more specific interpretation below.

- *Classification of different software models, measuring the size of different software models, and measuring of the amount of reuse between different software models requires modification of the dynamic part of the meta-model architecture that classifies different software model types*[22] **(outcome 3)**

A different software model type classification needs to be added for each different software model type. Results indicate that the software model type classification methodology phase was the only phase that contained test cases where meta-level components were added or modified. No meta-level components were added or modified during the software model classification or measurement testing methodology phases. The addition of meta-level components occurs for each test case in the software model type classification methodology phase. Modification of meta-level components was a result of copying and modifying some software model type classifications to exploit similarities between different software model types, that is, reuse between different software model type classifications.

- *The static part of the meta-model architecture and the automation specification was not changed to classify software models, measure their size, or measure the amount of reuse with different software models or software model types, including object-oriented or UML software model types*[23] **(outcome 4 and 5).**

Although some limitations were identified in previous phases, the measurement framework and its implementation remained stable with no changes required to account for different software model types, object-oriented software model types, or UML software model types. More specifically, changes to chapter three and additional coding and re-compilation of the software for the prototype tool were not required to account for different software model types, object-oriented software model types, or UML software model types. However, findings from the measurement testing methodology phase for composition reuse was not completely successful. To change the outcome it may be necessary to change the static part of the

---

[22] (SMTC_MLCA_LI = 39, SMTC_MLCM_LI = 19, OU_SMTC_MLCA_LI = 26, OU_SMTC_MLCM_LI = 14), (SMC_MLCA_LI = 0, SMC_MLCM_LI = 0, MT_MLCALI = 0, MT_MLCM_LI = 0, OU_SMC_MLCA_LI = 0, OU_SMC_MLCM_LI = 0, OU_MT_MLCALI = 0, OU_MT_MLCM_LI = 0).

[23] Hypothesis measures with the suffix _ASPEC_LI, _ASPTC_LI, _MMAPEC_LI, and _MMAPTC_LI all equal 0.

meta-model architecture and the automation specification. If this was done during the measurement testing methodology phase then the indicators for modification of the static part of the meta-model architecture and automation specification would be very different.

The discrepancy between results for all software model types in Table 5-11 and OO/UML software model types in Table 5-12 is due to the set of hypothesis measures that were used for all software model types but cannot be used specifically for OO/UML software model types.[24] These ten indicators cannot be applied to a subset of software model types such as object-oriented or UML software model types. Apart from this, summary analysis is identical for all software model types and OO/UML software model types

With regard to **outcome 2**, if the software model classification methodology phase was split into two phases, one for classification of software models and the other for measurement of their size, then another finding would have been evident. That is, the need to have a specialised meta-level component for measurement separate from an meta-level component for classification of software models. This cannot be demonstrated because tests for size and classification of software models were in the same phase.

With regard to **outcome 2**, other results that support H2 are due to hypothesis measures for usage of the amount of reuse measurement M2 model specifier MLC (ARMM2MS). The results for any usage indicator with this MLC is always 0 (for example, see Table 5-14, Table 5-15, Table 5-15, and Table 5-16 for this MLC). It was difficult to identify instances of the MLC. The ARMM2MS MLC contains the definitions of the software model type set theory MLC, the software model type classification MLC, and the amount of reuse measurement M1 model specifier MLC. These three MLCs have instances, but what are the instances of the ARMM2MS? For this reason it cannot be said that this MLC was used in any phase, at least not directly through its instances. Perhaps it may be labelled an "abstract MLC", that is, an MLC that has no instances. This is revisited in chapter 6 as a further research issue.

---

[24] MINCOT_LI, MAXCOT_LI, MEACOT_LI, MEDCOT_LI, MODCOT_LI, MINEC_LI, MAXEC_LI, MEAEC_LI, MEDEC_LI, and MODEC_LI.

With regard to **outcome 3**, it could be argued that addition of instances for meta-level one meta-level components points to a limitation. However, it is assumed that:

- Addition of meta-level one instances constitutes data entry.
- The component of data entry in any software tool does not constitute a unique limitation.

There are still some significant issues related to addition of meta-level one MLC instances that are considered in chapter 6.

The next section analyses the degree of support for the different hypotheses based on results for hypothesis measures in the hypothesis analysis reports.

5.3 Analysis by Hypothesis

Results from the methodology phases are also duplicated for each hypothesis. This section examines the impact of results on hypotheses. The contributions of each methodology phase to analysis of hypotheses are manifest in the hypothesis analysis reports. These reports are contained in Appendix D.

H0a/b

Details of results for hypothesis H0 can be found in the H0 hypothesis analysis report in Appendix D5. A guide to interpretation of measures can be found in Appendices C2 – C5.

Table 5-18 summarises the results for the null hypothesis H0. Examination of this table shows that the hypothesis is denied by virtue of indicators for any hypothesis part.

Table 5-18: Results for Hypotheses H0a/b

Hypothesis Part	Number of Indicators	Number That Support	Percentage That Support	Number that Deny	Percentage That Deny
H0a All SMT	20	0	0%	20	100%
H0a OO/UML SMT	20	0	0%	20	100%
H0b All SMT	100	43	43%	57	57%
H0b OO/UML SMT	90	40	44%	50	66%
TOTAL H0a/b	**230**	**83**	**36%**	**147**	**64%**

The extract for H0a states that

> "...A framework based on meta-modelling... Cannot support measurement of the amount of reuse for any kind of software model..."

Examination of results for H0a from the methodology phases indicate that:

1. At least one kind of software model type was classified *(results from software model type classification)*.
2. At least one software model was successfully classified and measurement of size was also successful for at least one software model type *(results from software model classification)*.
3. Measurement of the amount of reuse was successful for at least one software model and one software model type *(results from measurement testing)*.
4. At least one kind of object-oriented or UML software model type was classified *(results from software model type classification)*.
5. At least one object-oriented or UML software model was successfully classified and measurement of size was also successful for at least one object-oriented or UML software model type *(results from software model classification)*.
6. Measurement of the amount of reuse was successful for at least one object-oriented or UML software model and one object-oriented or UML software model type *(results from measurement testing)*.

Results for H0a contradict the extract for H0a. Hence, H0a is denied due to results for H0a.

A Significant Point: there is no fundamental difference in results for object-oriented/UML software model types or software model types in general.

The extract for H0b states that

> "...A framework based on meta-modelling... Does not have any limitations in measurement of the amount of reuse with different kinds of software models..."

Examination of results for H0b from the methodology phases indicate that:

1. There were limitations with classifying different software model types *(results from software model type classification)*.
2. There were some limitations with classifying and measuring the size of different software models and software model types *(results from software model classification)*.
3. There were some limitations with measuring the AOR of different software models and software model types *(results from measurement testing)*.
4. Limitations were discovered for measurement of the AOR with different software model types using an automated version of the measurement framework *(results from automation assessment)*.
5. There were limitations with classifying different object-oriented or UML software model types *(results from software model type classification)*.
6. There were some limitations with classifying and measuring the size of different object-oriented or UML software models and software model types *(results for software model classification)*.
7. There were some limitations with measuring the AOR of different object-oriented or UML software models and software model types *(results for measurement testing)*.
8. Limitations were discovered for measurement of the AOR with different object-oriented and UML software model types using an automated version of the measurement framework *(results for automation assessment)*.

Results for H0b contradict the extract for H0b. Hence, H0b is denied due to results for H0b.

Table 5-18 indicates that not all indicators for H0b could deny H0b. This is in contrast to results for H0a where all indicators were able to deny H0a. Even with these variations H0a and H0b remain null and void. Thus, it can be argued that the following statements (**H0a/b**) are **not supported** by results of the methodology phases.

A framework based on meta-modelling:

- A framework based on meta-modelling cannot support measurement of the amount of reuse for any kind of software model.
- A framework based on meta-modelling does not have any limitations in measurement of the amount of reuse with different kinds of software models.

This statement is a copy of the null hypotheses H0a and H0b, respectively.

H1

Table 5-19 summarises the results for the hypothesis H1 for all software model types. Examination of this table shows that the hypothesis has support, based on results for each sub-hypothesis. Table 5-20 summarises the results for the hypothesis H1 for object-oriented and UML software model types. Examination of this table shows that the hypothesis has support, based on results for each sub-hypothesis.

Table 5-19: Results for Hypothesis H1 for all software model types.

Sub-Hypothesis	Number of Indicators	Number That Support	Percentage That Support	Number That Don't Support	Percentage That Don't Support
SH 1.1	13	13	100%	0	0%
SH 1.2	5	5	100%	0	0%
SH 1.3	5	5	100%	0	0%
SH 1.4	5	5	100%	0	0%
SH 1.5	10	10	100%	0	0%
SH 1.6	10	10	100%	0	0%
SH 1.7	10	10	100%	0	0%
SH 1.8	10	10	100%	0	0%
SH 1.9	10	10	100%	0	0%
SH 1.10	10	10	100%	0	0%
SH 1.11	10	10	100%	0	0%
SH 1.12	10	10	100%	0	0%
TOTAL H1	**108**	**108**	**100%**	**0**	**0%**

Table 5-20: Results for Hypothesis H1 for OO/UML software model types.

Sub-Hypothesis	Number of Indicators	Number That Support	Percentage That Support	Number That Don't Support	Percentage That Don't Support
SH 1.1	13	13	100%	0	0%
SH 1.2	5	5	100%	0	0%
SH 1.3	5	5	100%	0	0%
SH 1.4	5	5	100%	0	0%
SH 1.5	10	10	100%	0	0%
SH 1.6	10	10	100%	0	0%
SH 1.7	10	10	100%	0	0%
SH 1.8	10	10	100%	0	0%
SH 1.9	10	10	100%	0	0%
SH 1.10	10	10	100%	0	0%
SH 1.11	10	10	100%	0	0%
SH 1.12	10	10	100%	0	0%
TOTAL H1	**108**	**108**	**100%**	**0**	**0%**

The extract for H1 states that:

> "...A framework based on meta-modelling supports measurement of the amount of reuse for different kinds of software models...."

Examination of results for H1 for all software model types from the methodology phases indicate that:

1. It is possible to classify different software model types *(results for SH 1.1 from software model type classification).*

2. It is possible to classify and measure the size of different software models based on their type using the measurement framework *(results for SH 1.1 - 1.4 from software model classification).*
   More specifically,

   a) If a software model type can be classified then it is possible to classify software models of the same type and measure their size for the purposes of measurement of the amount of reuse.

3. It is possible to measure internal composition reuse, external composition reuse, internal generation reuse, and external generation reuse for different software models and different software model types *(results for SH 1.5 - 1.12 from measurement testing).*
   More specifically,

   a) If a software model type can be classified then it is possible to measure the amount of reuse with a high degree of success.

   b) Successful software model type classification appears to guarantee successful measurement of generation reuse.

Results for all software model types support the extract for H1. Hence, H1 is supported due to results for its sub-hypotheses. Results indicate that successful classification of the software model type is needed for successful measurement of the amount of reuse.

Examination of results for H1 for object-oriented and UML software model types from the methodology phases indicate that:

1. It is possible to classify different object-oriented and UML software model types *(results for SH 2.2 from software model type classification).*

2. It is possible to classify and measure the size of different object-oriented and UML software models based on their type using the measurement framework *(results for SH 2.2 - 2.4 from software model classification).*
   More specifically,

   a) If a software model type can be classified then it is possible to classify software models of the same type and measure their size for the purposes of measurement of the amount of reuse.

3. It is possible to measure internal composition reuse, external composition reuse, internal generation reuse, and external generation reuse for different object-oriented and UML software models and different object-oriented and UML software model types *(results for SH 2.5 - 2.12 from measurement testing).*
   More specifically,

   a) If a software model type can be classified then it is possible to measure the amount of reuse with a high degree of success.

   b) Successful software model type classification appears to guarantee successful measurement of generation reuse.

Results for object-oriented and UML software model types support the extract for H1. Hence, H1 is supported due to results for its sub-hypotheses. Again, successful classification of the object-oriented or UML software model type is needed for successful measurement of the amount of reuse. This is similar to the requirement identified for all software model types above.

It can still be argued that the following statement (**H1**) is ***supported*** by results from the methodology phases.

A framework based on meta-modelling can support measurement of the amount of reuse for different kinds of software models. In particular,

- A framework based on meta-modelling can classify different kinds of software models.
- A framework based on meta-modelling can classify different software models based on their type.
- A framework based on meta-modelling can measure size for different kinds of software models.
- A framework based on meta-modelling can measure size for different software models.
- A framework based on meta-modelling can measure the amount of reuse for internal composition reuse for different kinds of software models.
- A framework based on meta-modelling can measure the amount of reuse based on external composition reuse for different kinds of software models.
- A framework based on meta-modelling can measure the amount of reuse for internal generation reuse for different kinds of software models.
- A framework based on meta-modelling can measure the amount of reuse for external generation reuse for different kinds of software models.
- A framework based on meta-modelling can measure the amount of reuse for internal composition reuse for different software models.
- A framework based on meta-modelling can measure the amount of reuse for external composition reuse for different software models.
- A framework based on meta-modelling can measure the amount of reuse for internal generation reuse for different software models.
- A framework based on meta-modelling can measure the amount of reuse for external generation reuse for different software models.

This statement is a copy of the hypothesis H1 and its sub-hypotheses.

H2

Table 5-21 summarises the results for the hypothesis H2 for all software model types. Examination of this table shows that the hypothesis has support, based on results for each sub-hypothesis. Table 5-22 summarises the results for the hypothesis H2 for object-oriented and UML software model types. Examination of this table shows that the hypothesis has support, based on results for each sub-hypothesis.

Table 5-21: Results for Hypothesis H2 for all software model types

Sub-Hypothesis	Number of Indicators	Number That Support	Percentage That Support	Number That Don't Support	Percentage That Don't Support
SH 2.1 - 2.12	68	35	51%	33	49%
SH 2.1	60	36	60%	24	40%
SH 2.2	26	5	19%	21	81%
SH 2.3	56	8	14%	48	86%
SH 2.4	22	2	9%	20	91%
SH 2.5	114	22	19%	92	81%
SH 2.6	114	22	19%	92	81%
SH 2.7	114	20	18%	94	82%
SH 2.8	114	20	18%	94	82%
SH 2.9	44	8	18%	36	82%
SH 2.10	44	8	18%	36	82%
SH 2.11	44	6	14%	38	86%
SH 2.12	44	6	14%	38	86%
TOTAL H2	**864**	**198**	**23%**	**666**	**77%**

Table 5-22: Results for Hypothesis H2 for object-oriented and UML software model types

Sub-Hypothesis	Number of Indicators	Number That Support	Percentage That Support	Number That Don't Support	Percentage That Don't Support
SH 2.1 - 2.12	58	34	58%	24	42%
SH 2.1	60	32	53%	28	47%
SH 2.2	26	5	19%	21	81%
SH 2.3	56	8	14%	48	86%
SH 2.4	22	2	9%	20	91%
SH 2.5	114	21	18%	93	82%
SH 2.6	114	21	18%	93	82%
SH 2.7	114	19	16%	95	84%
SH 2.8	114	19	16%	95	84%
SH 2.9	44	8	18%	36	82%
SH 2.10	44	8	18%	36	82%
SH 2.11	44	6	14%	38	86%
SH 2.12	44	6	14%	38	86%
TOTAL H2	**854**	**189**	**22%**	**665**	**88%**

The extract for H2 states that:

> "...A framework based on meta-modelling has limitations in measurement of the amount of reuse with different kinds of software models..."

Examination of results for H2 for all software model types from the methodology phases indicate that:

1. Classification of different software model types has limitations *(results for SH 2.1 from software model type classification)*.
   More specifically,
   a) Out of 55 software model types only 40 were linked to implementations and used for testing the measurement framework.
2. There are some limitations identified in the measurement framework when classifying and measuring size of different software models and different kinds of software models *(results for SH 2.2 - 2.4 from software model classification)*.
   More specifically,
   a) A number of software model types remain untested with respect to classification and measurement of size.
   b) If any two software models are based on a different software model type, then different software model type classifications are required for successful classification of these software models and measurement of their size.

3. There are some limitations identified in the measurement framework when measuring the amount of internal composition reuse, external composition reuse, internal generation reuse, and external generation reuse with different software models and software model types *(results for SH 2.5 - 2.12 from measurement testing)*.
   More specifically,

   a) A number of software model types remain untested with respect to measurement of reuse.

   b) Successful software model type classification does not guarantee successful measurement of reuse.

   c) Successful software model type classification does not guarantee successful measurement of composition reuse.

   d) If any two software models are based on different software model types, then different software model type classifications are required for successful measurement of reuse for their respective software models.

4. There is a limitation of the framework related to the automation specification and use of meta-level components across different phases with different software model types *(results for SH 2.1 - 2.12 from automation assessment)*.
   More specifically,

   a) A degree of specialisation is required for classification of different software model types prior to classification of software models, measurement of their size, and measurement of the amount of reuse. Only three meta-level components were used to support classification of different software model types.

   b) A degree of specialisation is also required for classification of software models, measurement of size, and measurement of the amount of reuse. Four meta-level components were used only for classification of software models, measurement of size, and measurement of the amount of reuse. However, the same set of meta-level components were used for classification of software models, measurement of size, and measurement of the amount of reuse with different software model types and different software models.

5. There is a limitation with classification of different software model types *(results for SH 2.1 from automation assessment)*.
   More specifically,

a) Measurement of the amount of reuse requires modification of the dynamic part of the meta-model architecture that classifies different software model types. A different software model type classification needs to be added for each different software model type.

6. There are few limitations identified for classification of software models, measurement of their size, and measurement of the amount of reuse for different approaches and different kinds of software models *(results for SH 2.2, SH 2.3, SH 2.4, SH 2.5, SH 2.6, SH 2.7, SH 2.8, SH 2.9, SH 2.10, SH 2.11, SH 2.12 from automation assessment).*
More specifically,

a) It is unlikely that a separate set of code in the automation specification was used to classify different software models to measure the amount of reuse for each software model type.

b) The static part of the meta-model architecture and the automation specification did not require any change to classify software models, measure their size, or measure the amount of reuse with different software models or software model types.

Results for all software model types support the extract for H2. This is sufficient to support H2. Note that some indicators did not support H2. However, if research was done to address failures for measurement of composition reuse with implementation models then this would have further supported H2.

Examination of results for H2 for object-oriented and UML software model types from the methodology phases indicate that:

1. Classification of different object-oriented or UML software model types has limitations *(results for SH 2.1 from software model type classification)*. More specifically,

    a) Out of 35 object-oriented or UML software model types only 26 were linked to implementations and used for testing the measurement framework.

2. There are some limitations identified in the measurement framework when classifying and measuring size of different object-oriented and UML software models *(results for SH 2.2 - 2.4 from software model classification)*. More specifically,

    a) A number of software model types remain untested with respect to classification and measurement of size.

    b) If any two software models are based on different software model types, then different software model type classifications are required for successful classification of these software models and measurement of their size.

3. There are some limitations identified in the measurement framework when measuring the amount of internal composition reuse, external composition reuse, internal generation reuse, and external generation reuse with different object-oriented and UML software models and software model types *(results for SH 2.5 - 2.12 from measurement testing)*.
   More specifically,
   a) A number of object-oriented and UML software model types remain untested with respect to measurement of reuse.
   b) Successful object-oriented or UML software model type classification does not guarantee successful measurement of reuse.
   c) Successful object-oriented or UML software model type classification does not guarantee successful measurement of composition reuse.
   d) If any two software models are based on different object-oriented or UML software model types, then different software model type classifications are required for successful measurement of reuse for their respective software models.
4. There is a limitation of the framework related to the automation specification and use of meta-level components across different phases with different object-oriented and UML software model types *(results for SH 2.1 - 2.12 from automation assessment)*.
   More specifically,
   a) A degree of specialisation is required for classification of different object-oriented or UML software model types prior to classification of software models, measurement of their size, and measurement of the amount of reuse. Only three meta-level components were used to support classification of different software model types.
   b) A degree of specialisation is also required for classification of object-oriented or UML software models, measurement of size, and measurement of the amount of reuse. Four meta-level components were used only for classification of software models, measurement of size, and measurement of the amount of reuse. However, the same set of meta-level components were used for classification of software models, measurement of size, and measurement of the amount of reuse with different object-oriented and UML software model types and different software models.

5. There is a limitation with classification of different object-oriented and UML software model types *(results for SH 2.1 from automation assessment)*.
   More specifically,

   a) Measurement of the amount of reuse requires modification of the dynamic part of the meta-model architecture that classifies different object-oriented or UML software model types. A different software model type classification needs to be added for each different software model type.

6. There are few limitations identified for classification of software models, measurement of their size, and measurement of the amount of reuse for different approaches and different kinds of object-oriented and UML software models *(results for SH 2.2, SH 2.3, SH 2.4, SH 2.5, SH 2.6, SH 2.7, SH 2.8, SH 2.9, SH 2.10, SH 2.11, SH 2.12 from automation assessment)*.
   More specifically,

   a) It is unlikely that a separate set of code in the automation specification was used to classify different software models to measure the amount of reuse for each object-oriented or UML software model type.

   b) The static part of the meta-model architecture and the automation specification did not require any change to classify software models, measure their size, or measure the amount of reuse with different object-oriented or UML software models or software model types.

Results for object-oriented and UML software model types are very similar to results for all software model types. Results for object-oriented and UML software model types support the extract for H2. This is sufficient to support H2. Some indicators did not support H2. If research was done to address failures for measurement of composition reuse with implementation models then this would have further supported H2.

In any case the following statement (**H2**) is ***supported*** by results from the methodology phases.

A framework based on meta-modelling has limitations in measurement of the amount of reuse with different kinds of software models. In particular,

- A framework based on meta-modelling that can classify different kinds of software models has limitations.
- A framework based on meta-modelling that can classify different software models based on their type has limitations.
- A framework based on meta-modelling that can measure size for different kinds of software models has limitations.
- A framework based on meta-modelling that can measure size for different software models has limitations.
- A framework based on meta-modelling that can measure the amount of reuse for internal composition reuse for different kinds of software models has limitations.
- A framework based on meta-modelling that can measure the amount of reuse based on external composition reuse for different kinds of software models has limitations.
- A framework based on meta-modelling that can measure the amount of reuse for internal generation reuse for different kinds of software models has limitations.
- A framework based on meta-modelling that can measure the amount of reuse for external generation reuse for different kinds of software models has limitations.
- A framework based on meta-modelling that can measure the amount of reuse for internal composition reuse for different software models has limitations.
- A framework based on meta-modelling that can measure the amount of reuse for external composition reuse for different software models has limitations.
- A framework based on meta-modelling that can measure the amount of reuse for internal generation reuse for different software models has limitations.
- A framework based on meta-modelling that can measure the amount of reuse for external generation reuse for different software models has limitations.

This statement is a copy of the hypothesis H2 and its sub-hypotheses.

5.4 Summary of Results

Table 5-23 summarises the results for hypotheses H0a – H2.

Table 5-23: Results Summary for Hypotheses H0a – H2

Hypothesis	Number of Indicators	Support #; Percentage	Not Support #; Percentage	Deny #; Percentage
H0a/H0b	230	83; 36%		147; 64%
H1 All SMT	108	108; 100%	0; 0%	
H1 OO/UML SMT	108	108; 100%	0; 0%	
H2 All SMT	864	198; 23%	666; 77%	
H2 OO/UML SMT	854	189; 22%	665; 78%	

Table 5-23 illustrates a few major themes discovered for analysis of hypotheses.

1. Although some indicators did support H0a and H0b, others did not and this is sufficient to deny the null hypotheses H0a and H0b.

2. The summary of results for H1 using all software model types versus object-oriented and UML software model types. This suggests that there are no fundamental differences in success or failure of measurement for the amount of reuse with software model types when compared to object-oriented and UML software model types.

3. The summary of results for H2 are very close for all software model types when compared to results for object-oriented and UML software model types. This suggests that there is no fundamental difference in limitations of the measurement framework for different software model types when compared to object-oriented or UML software model types.

4. When comparing results for H1 to H2 the level of support for H1 is much higher than H2. This does not mean that the limitations identified are insignificant. In particular, ten specific limitations were listed for H2 in their respective sections (see points 1a), 2(a), 2(b), 3(a), 3(b), 3(c), 3(d), 4(a), 4(b), and 5(a) under the heading H2 in this chapter).

This concludes data analysis for methodology phases and hypotheses. The next chapter summarises they key findings of this thesis, its main contribution to knowledge, and issues for further research.

Chapter 6 Conclusions

This chapter is divided into four parts. The first part consists of the main contributions'. This part seeks to describe the most significant contributions to knowledge made by this thesis. The second part is key findings. This part summarises the main findings arising directly from the results of the assessment framework. The third part is discussion. This part describes some research issues without any explicit research questions. The fourth part is further research. This part covers areas for improvement to the measurement framework and areas outside the limitations of this thesis that deserve attention. The fourth part includes a range of new research questions.

6.1 Main Contributions

Three main contributions of this thesis are identified:

1. A Single Measurement Theory for the Amount of Reuse. This work has shown that development of a set of measures can be applied to a range of different software model types. In this way meaningful comparisons can be made to assess the influence of different software processes on software reuse practice. This was achieved using meta-modelling and set theory.

2. A Substantial Contribution to Measures for The Amount of Reuse. In particular, three areas that were in need of refinement were addressed in this work:

a) **Measures were defined for analysis and design models.** It could be argued that this work does not have a significant contribution due to failure of accurate measurement for implementation models (source code). The author responds to this by citing ***[1] "...Until we develop a comprehensive system of measuring and evaluating reuse in each of the individual software life-cycle phases, code reuse metrics will remain our best method of assessing the overall benefits of reuse..." (p. 569)***. Even if the measurement framework is considered unacceptable as a general theory and even with the failure to accurately measure composition reuse with implementation models, the impact of the work remains. Measures were developed to evaluate reuse for the analysis and design phases of the software life cycle and this can be used as another means of assessing the benefits of reuse.

b) **Measures were defined for generation reuse.** According to [2] and a subsequent literature review in chapter 2 of this thesis (Section 2.5), this was a weak area in need of refinement.

c) **Refinement of two indicators for the amount of reuse.** These are *waste generated* for generation reuse, and *amount not reused* for composition reuse. Although [3] mention the notion of a poor reuse record, few authors discuss or mention two measures introduced in this work.[1]

3. Demonstration of the Practical Application of a Theory Base for Meta-modelling. This thesis has demonstrated that a refined approach to meta-modelling can be used to good effect at solving a research problem. In particular, in measurement of software reuse.

---

[1] Waste generated and amount not reused are not mentioned in the following sources [Frakes, 1996 #510; Fenton, 1991 #499; Chidamber, 1991 #381; Chidamber, 1994 #479; Basili, 1996 #504; Banker, 1994 #508; Hitz, 1995 #553]. In addition, the literature review for this in chapter two states that only 3 sources addressed amount not reused and two sources addressed waste generated.

6.2 Key Findings

The key findings from the assessment framework are:

1. Hypotheses H1, H2, H3, and H4 were supported. The null hypothesis H0 was denied.

2. There were no fundamental differences between results for object-oriented and UML software model types (H2, H4) and results for software model types in general (H1, H3):

   a) For the level of success in application (H1, H2)

   b) For identification of any limitations (H3, H4)

3. The measurement framework was effective for measurement of:

   a) Composition reuse with analysis and design models.

   b) Generation reuse.

4. The measurement framework was not effective for measurement of composition reuse with implementation models.[2] To make the measurement framework effective for this may have required modification of the static part of the meta-model architecture and automation specification. This would have led to identification of more limitations than were found with the current measurement framework.

5. To have successful measurement of the amount of reuse using the measurement framework:

   a) A software model type must be classified using the appropriate MLCs in the measurement framework.

   b) Each software model type must have a different classification scheme. This requires additions to the dynamic part of the meta-model architecture in the measurement framework. However, this has always been a limitation of meta-CASE tools, and it does not involve re-coding and re-compilation of the tool.

---

[2] The measures for reuse of implementation models tended to be overstated and a rule of thumb had to be applied to make them more realistic. This rule is "If the code compiles without it, it was not reused". In any case this was a tedious process that could not be reduced to a routine procedure of classifying the software model based on its software model type classification. For this reason, implementation models were not successful and it is likely that any successful test case is due to the effort of the researcher and not the measurement framework.

6. A degree of specialisation in the measurement framework was required to separate classification of software model types from classification of software models, measurement of size, and measurement of the amount of reuse. Further specialisation may have been found if the software model classification methodology phase was separated into two phases, one for classification of the software models and the other for measurement of their size.

6.3 Discussion

[4] put forward the following calculation for internal reuse percent.

INTERNAL REUSE PERCENT = 100 - NEW OBJECT PERCENT - EXTERNAL REUSE PERCENT.

The nearest equivalent using the measurement framework is

INTERNAL REUSE PERCENT = 100 - Percent Amount Added - Percent Amount Reused.

Using the measurement framework, this number is always zero. This is because the measurement framework uses ownership of the software model as the basis for measurement of internal and external reuse. This was also stated as a limitation of the study (see chapter 1, LN-3). However, a significant problem is related to the interpretation of data that the measurement framework was designed to support. If a model element is developed and reused within the same project it does not necessarily decrease the amount of key tapping required to develop an application. The range of propositions for measurement of internal reuse[3] may need to consider to the interpretation of data they wish to support. For this reason the measurement framework adopts the view that a software mode reused in another software model yields some clear division between work done and work reused.

The identifier parts can vary in length from a setting of about 5 characters to a line of text of up to 40 characters and a line number. This indicates a great deal of variation in the amount of key tapping that is not recognised in the measurement framework. Based on the interpretation of data perhaps this should be incorporated using the number of characters for each model element as a multiplier. This would probably require a change in the set theory for measurement.

[3] [Fenton, 1991 #499] defines private reuse as reuse that occurs within a product. [Banker, 1994 #508] interpret internal reuse as reused objects, not counting the first occurrence of an object written within the same project. [Frakes, 1996 #510] calculate internal reuse as the number of items not from an external repository used more than once. [Frakes, 1996 #510] also put forward the concept of counting references to items (Model concepts) for reuse frequency.

The measurement framework is not a complete success. It could be argued that a more comprehensive solution should have been devised. There is a counter argument to devising a more comprehensive solution if we assume that a more comprehensive solution is a more complex system. [5] states that:

"A complex system that works is invariably found to have evolved from a simple system that worked... A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over, beginning with a working simple system..."

This can be interpreted as:

"If you build a complex system from a simple system the complex system will work."

"If you build a complex system from scratch it will not work."

We agree with the above and from this it can be said that:

- The measurement framework is a system that works within certain limits and
- The measurement framework can be used to support better solution using a more complex system.

Is there anything that may suggest the cause of failure of the measurement framework for measuring the amount of reuse with implementation models? One feature that appears to distinguish implementation models from analysis and design models is that of graphic representation versus text. Is it a coincidence that the graphic representation models are easier to classify and have better results compared to text representation models? One software model type had a graphic representation but was very close to the implementation (OEW, SMTI ID: 21). This software model type still had good results. Perhaps the difference of results with the implementation models versus analysis/design models is in part to do with their representation and how these models are developed using CASE technology.

One observation made about UML is based on the Paradigm Plus CASE tool. Based on this implementation and UML semantics [6] it appears that UML allows for a large amount of flexibility in modelling.[4] The author could not find a text book that made use of this flexibility.[5] Further examination of UML semantics [6] support the interpretation inferred from the Paradigm Plus CASE Tool. For this reason such flexibility may not be due to a choice made by the Vendors of the Paradigm Plus CASE tool. However, it seems that the UML standard is not as rigorous as previous modeling approaches when compared to different CASE tools.[6] Why does UML semantics allow for so much flexibility when it is not used in text books? Moreover the UML model appears to be a flat model, in contrast to arguments made by [7]. In addition, traceability from analysis to design as described by [8] (p. 439 - 442) cannot be supported using UML as described and implemented. How can an analysis model be traced to a design model when they are not separately defined on a CASE tool?

---

[4] This includes:

1. Classes appearing in Use Case Diagrams.
2. Collaboration and Object Diagrams containing an identical set of model element types and notation, even though a document included in the tool called "UML Notation" describes them separately.
3. Classes appearing in state diagrams.
4. States containing attribute values and operations. (Why not actions?)
5. Components containing any diagram you like.
6. Modules, subsystems, and class category used a means of distinguishing the role and appearance of a component only.

[5] Sources include a number of texts [Fowler, 1997 #745] [Schach, 1999 #807] [Jacobson, 1997 #619; Jacobson, 1999 #818] [Booch, 1999 #900], and the standards documents [Rational, 1997 #740; Rational, 1997 #739; Rational, 1997 #852; Rational, 1997 #736; Rational, 1997 #141]

[6] Two good examples for comparison are the Rational Rose C++ CASE tool and the WithClass Case tool.

Use of meta-modelling and set theory appears to support candidate solutions to other problems for measurement of reuse. For example, what if a library model is associated with low values for amount not reused. Is this a poor library model? Perhaps different parts of it are being reused. Use of set theory leads to the following proposition.

$$\text{Aggregate Reuse History} = |\,(M1 \cup M2 \ldots \cup Mn) \cap \text{LibraryModel}\,|.$$

All of the elements in various models (M1 .. Mn) are gathered and compared to the Library Model.

There was an MLC in the measurement framework that did not have any instances of itself. The MLC was the Amount of Reuse Measurement M2 Model Specifier. The MLC contained two other MLCs. To adequately classify this MLC may require the introduction of an abstract MLC as a concept in meta-modelling. Abstract MLCs do not have any instances.

This work does support the theme of convergence in [9]. Looking at semantic space in [10](pp. 524 - 526), there is use of set theory to distinguish between different concepts in software model types. [11] (Figure 1, page 88) illustrate the notion of sematic correspondence for different kinds of software models. This diagram looks like a Venn diagram with areas representing different kinds of software models and overlap representing correspondence between them.

To test the notion of convergence, preliminary experiments for measurement of the amount of reuse were done using different software model types. For example, a class diagram as a library model and a state diagram as an application model. The actual results were equal to the expected results. Other experiments were done to create software model types with different names and different overall structure but with some common structural parts similar to common sematic correspondence in [11]. The actual results were also equal to the expected results. What implications does this have for software reuse practice?

6.4 Further Research

To support specification of the measurement framework two diagrams were introduced in Appendix C2.[7] In contrast to other architectural styles in software engineering, the meta-model architectural style was supplemented with separate diagrams for specifying the architecture of the measurement framework. Other architectural styles do not require a separate diagram for the architecture of a system[8] but there are reasons to support a more formal approach to specification of a software architecture based on its style [12]. This leads to the following questions:

1. Do the meta-model diagrams constitute a contribution to software architecture design?
2. If so, what are the implications for software architecture design?

[7] These were the meta-model architecture diagram and the meta-model layer diagram.

[8] For example, a UML architecture diagram: in most cases the software architecture is manifest in a draft of the complete design specifications.

[13] (p. 157 - 160)[9] describes a problem with measurement of size for analysis and design models that was not resolved at the time. More recently [14] (p. 16 - 17)[10] describes a similar problem of comparison between different analysis and design software model types as unresolved for the current set of complexity metrics. This thesis has demonstrated that measurement of size and the amount of reuse can be obtained for different analysis and design software model types. [15] (p. 315, 318) contrasts the ease of measuring reuse with text based models with the difficulty of measuring reuse for graphic based models and higher level models (analysis, design, etc.).[11] One component of the measurement framework included representation of any software model type and its software model using text. Software model types, such as class diagrams and data flow diagrams, can be exported as text models. This leads to the following questions:

1. Could the measurement framework be used to solve the problem in [13]?
2. Could the measurement framework be used to solve the problem in [14]?
3. Could the measurement framework be used to solve the problem in [15]?

---

[9] The following quotes are from [Fenton, 1991 #499]. "... The state-of-the-art for size measurement is that... there is some consensus view on measuring length of programs but not if specifications or designs..." (p. 157). "...Defining... length measures for specification and design documents is, unfortunately, not so easy... such documents consist of a myriad of text, graphs, and special mathematical diagrams and symbols. The nature of these will depend on the particular style, method, or notation used..." (p. 159 - 160)

[10] The following quotes are from [Whiddett, 1997 #579]. "...The major problems encountered during this study arose from the unavailibility of a suitable tool to measure the complexity of different systems and which could cope with the different constructs and formulations of the models. The most important need that is indicated from this research is to investigate ways of making complexity measurements that can be applied to a wide variety os systems..." (p. 16 - 17). "...An appropriate metric needs to be able to take into account the unique differences of both the structured and object-oriented methodologies..." (p. 17).

[11] From [Leach, 1996 #1090] "...The same transformation can apply to any software artefact given in textual format, such as requirements, designs in PDL, test plans, test results and documentation..." (p. 315), and "...Measuring the amount of reuse in higher level software artefacts is more difficult than measuring reuse of source code. The wide range of design representations (test-based PDL, graphics-based flowcharts, data flow diagrams, or other diagrams CASE tools, etc.) make automatic collection of reuse information difficult..." (p. 318)

A number of sources discuss reuse within a product (that is, a software model) based on some kind of reference to a part of the product (that is, a model element).[12] Distinguishing between references to model elements and definition of model elements is not in the measurement framework. This may be the reason why measuring the amount of reuse using composition is not effective for implementation models. This leads to the following questions:

1. To account for references, what modifications to the measurement framework are needed?
2. What impact do these changes have on measurement of the amount of reuse?

Model element types from software model types were reused to classify other software model types. In addition, software model types with a common subset of model element types and composition relationships were used to assist reuse and modification of test cases. How this was done and its implications were not explored. This leads to the following questions:

1. What impact does reuse of model element types have on:
   a) Software model types?
   b) Reuse of test cases for measurement of the AOR for software model types?
2. What are the most effective methods for reuse of model element types?
3. How should the measurement framework be modified to facilitate reuse of model element types?

[12] [Fenton, 1991 #499] defines private reuse as reuse that occurs within a product. [Banker, 1994 #508] interpret internal reuse as reused objects, not counting the first occurrence of an object written within the same project. [Frakes, 1996 #510] calculate internal reuse as the number of items not from an external repository used more than once. [Frakes, 1996 #510] also put forward the concept of counting references to items (Model concepts) for reuse frequency.

Some test cases for different software model types were generated with the assistance of various software tools. However, in an industrial environment this will not be the case. The software models need to be imported to be measured. No systematic and automated means of importing were found.[13] This leads to the following questions:

1. What methods can be used to import software models into the automated version of the measurement framework?
2. How can the different methods be automated?
3. How effective are the different methods for automated importing of software models?

A number of potential sources for a solution were identified.[14]

Classification of context data and the measurement process remained fixed. If this varies with different organisations then this will have an impact on formation of measurement models for the amount of reuse. This leads to the following questions:

1. How could a measurement framework incorporate different classification schemes for collection, analysis, and interpretation of data?
2. What impact does this have on measurement of the amount of reuse?
3. What impact does this have in the process of measurement for the amount of reuse?

[13] There is a data entry interface to the prototype tool, but this is still a labour intensive process subject to some errors. Some experiments were done that include use of scripting languages and construction of translators specific to each software model type. This was assisted with the use of informal mappings of software model types to their respective classifications based on the measurement framework. A text importer facility for software models and software model types was made based on the software model type classification and software model classification MLCs but a CASE tool is needed to generate the text file using the correct grammar (that is, the TDL definition for the MLC).

[14] Alternate sources for automation of importing software models include

1. CASE integration standards (e.g. PCTE, CDIF, IRDS, OMG MOF)
2. Automatically generated compilers [Sloane, 1995 #503]
3. Re-targetable code generators [Ancona, 1995 #205]

A range of classification schemes for a software model type could be used but this was not explored.[15] In this thesis a choice was made based on what would most accurately reflect the amount of labour done to produce the software model. That is, how much of a software model was created by key strokes and how much was due to reuse of an existing software model. A problem of accuracy could occur in application. A classification scheme could be chosen and used before it is known that it is unsuitable. That is, measures do not reflect the interpretation of data for the practice of software reuse.[16] This leads to the following questions:

1. What effect do different classification schemes for a software model type have on
   - **a)** Results for Measurement of the AOR?
   - **b)** Quality of data for a given interpretation of it for the AOR?
2. How do different classifications schemes support different interpretations for the AOR?
3. How can classification schemes be made to support different interpretations of data for the AOR?

Some potential sources for a solution have been considered.[17]

---

[15] Different classification schemes can still work on the same data set but the richness of data can be affected. For example, Lisp can be classified using only lists and functions but classes are defined using functions. Classes in design are separate model element types. A classification scheme using only lists and not defining a separate model element types for classes means that reuse of classes cannot be detected.

[16] Variance of classification schemes for a software model type can have a varied impact on the values obtained for the amount of reuse. For example, assume that there is a class category in a library model and application model. Both of these contain the same classes and associations. If the user changes the name of the class category in the application model, none of the classes or associations are counted as contributions to the application mode. This can be alleviated by having classes and associations classified separately to class categories. Class categories are classified as containing references to classes and associations instead. On the other hand, if two classes contain an operation with a different name but a common parameter, this is also not counted. Is it likely that this parameter was somehow reused rather than typed in? Counting this similarity may be too optimistic.

[17] Five possible solutions to consider are:

1) Data is kept consistent and only one approach (conservative or optimistic) is used always.

In this thesis the M4 model and M3 model are not automated. They form part of a specification (definition). The automated version consists of the M2 model and M1 model (generation and interpretation). Based on the fundamentals of meta-modelling MLCs in meta-models are used to define other MLCs in other meta-models. This could be classified as follows:

- M4 -> M3 is definition.
- M3 -> M2 is definition and generation. M3 is used to generate M2. Similar to generation of code.
- M2 -> M1 is definition and interpretation. Similar to interpretation of code. M2 is used to interpret M1.
- M1-> M0 is definition and interpretation. M1 is used to interpret M0.

Using this scheme there appears to be a number of other combinations for definition, generation, and interpretation. These combinations are unexplored. This leads to the following questions:

1. What are the methods of automation between different meta-levels in a meta-model architecture?
2. What impact do these methods have on CASE tool construction to automate a meta-model architecture?

---

2) A range approach is adapted, in which reuse measurement is calculated using conservative and optimistic approaches. This would limit accuracy of predictions using the data for productivity based and size estimations.

3) Fuzzy logic [Giarratano, 1989 #527] is used. Fuzzy logic deals with the uncertainty of factors. such as whether or not a parameter definition from a class was reused. This places the above work in a good position because fuzzy logic relies on set theory.

4) Logging of CASE technology usage and correlating this with measurement data. For example, tracing the use of the keyboard to type in data rather than import a software model.

5) Use a representative sample of projects to conduct measures and test them against results for different classification schemes.

Meta-modelling is used in many areas related to software methodologies. After attendance at a lecture on meta-patterns based on [16] the author started to ponder the possibility of a meta-model architecture for construction of design patterns and an associated mechanism for generation of code based on these patterns. It became obvious that the experience gained from development of a meta-model architecture contributed to development and refinement of the initial specification. For example, the need to define a static counterpart to the MLCs in the dynamic part of the meta-model architecture. Methods of meta-model architecture design were not the focus of this work, but the author believes that valuable insights were gained through the application of meta-modelling in this work. A key recommendation is further research into application of meta-modelling to solving other research problems in software methodology areas. This leads to the following questions:

1. How could meta-modelling be used in other areas of software methodology research?
2. What contribution was made with the use of meta-modelling in these areas?

This concludes the research. The remaining sections contain the notes to this chapter, references, and further details that supplement the argument.

References

Chapter 1

No Citations

Chapter 2

1. Ghezzi, C., M. Jazayeri, and D. Mandrioli, *Software Engineering*. 1991, Englewood Cliffs, NJ: Prentice-Hall. 573.

2. McGregor, J.D. and D.A. Sykes, *Object-Oriented Software Development: Engineering Software for Reuse*. 1992, New York: Van Nostrand Reinhold. 351.

3. Layzell, P.J. and M.J. Freeman, *The Identification and Management of Latent Software Assets.* International Journal of Information Management, 1994. **14**(6): p. 427 - 442.

4. Henderson-Sellers, B. and J.M. Edwards, *Book Two of Object-Oriented Knowledge: The Working Object*. 1994, Englewood Cliffs New Jersey USA: Prentice-Hall.

5. Siegel, S., *Object-Oriented Software Testing: A Hierachical Approach*. 1996, New York: John Wiley & Sons. 511.

6. McClure, C., *CASE is Software Automtaion*. 1989, Englewood Cliffs, NJ: Prentice-Hall. 290.

7. McClure, C., *The Three Rs of Software Automation: Re-engineering, Repository, Reusability*. 1992, Englewood Cliffs, NJ: Prentice-Hall. 278.

8. Biggerstaff, T.J. and C. Richter, *Reusability Framework, Assessment, and Directions*, in *Software Reusability: Concepts and Models*. 1989, ACM Press-Addison Wesley: Reading, MA. p. 1 - 17.

9. Matsumoto, Y., *Some Experiences in Promoting Reusable Software*, in *Software Reusability: Applications and Experience*, T.J. Biggerstaff and C. Richter, Editors. 1989, ACM Press: Reading. p. 157 - 185.

10. Evans, M.W., *The Software Factory: A Fourth Generation Software Engineering Environment*. 1989, New York: John Wiley & Sons. 308.

11. Booch, G., *Object-Oriented Analysis and Design with Applications*. 2nd ed. 1994, Redwood City: Benjamin/Cummings. 589.

12. Frakes, W. and C. Terry, *Software Reuse: Metrics and Models.* ACM Computing Surveys, 1996. **28**(2): p. 415 - 435.

13. Jarzabek, S. *From reuse library experiences to application generation architectures.* in *Symposium on Software Reusability.* 1995. Seattle, Washington.

14. Cheng, J., *A Reusability-Based Software Development Environment.* Software Engineering Notes, 1994. **19**(2): p. 57 - 62.

15. NATURE, *Defining Visions in Context: Models, Processes, and Tools for Requirements Engineering.* Information Systems, 1996. **21**(6): p. 515 - 547.

16. Muhanna, W.A., *SYMMS: A model management system that supports model reuse, sharing, and integration.* European Journal of Operational Research, 1994. **72**: p. 214 - 243.

17. Mili, H., F. Mili, and A. Mili, *Reusing Software: Issues and Research Directions.* IEEE Transactions on Software Engineering, 1995. **21**(6): p. 528 - 562.

18. Fenton, N.E., *Software Metrics: A rigorous approach.* 1991, London: Chapman & Hall. 337.

19. Henderson-Sellers, B. and J.M. Edwards, *The Object-Oriented Systems Life Cycle.* Communications of the ACM, 1990. **33**(9): p. 142-159.

20. Stillings, N.A., *Cognitive Science: An Introduction.* 1987, Cambridge MA: MIT Press. 533.

21. Giarratano, J. and G. Riley, *Expert Systems: Principles and Programming.* 1989, Boston: PWS-KENT Publishing Co. 632.

22. Soley, R.M.E. and C.M. Stone, *Object Management Architecture Guide.* 3 ed. 1995, New York: Wiley & Sons. 164.

23. Margono, J. and T.E. Rhoads. *Software Reuse Economics: Cost-Benefit Analysis on a Large-Scale Ada Project.* in *14th International Conference on Software Engineering.* 1992: IEEE Computer Society Press.

24. Ramachandran, M. and A. Al-Yasiri. *Reusing and Retrieving Software Components: An Object-Oriented Domain Analysis Approach.* in *International Conference on Object-Oriented Information Systems.* 1994. London, UK: Springer-Verlag.

25. Karlsson, E.-A.E., *Software Reuse: A Holistic Approach.* 1995, Chichester UK: John Wiley & Sons. 510.

26. Sommerville, I., *Software Engineering.* 4th ed. 1992, Wokingham, England: Addison-Wesley. 649.

27. Banker, R.D., *et al.*, *Automating Output Size and Reuse Metrics in a Repository-Based Computer-Aided Software Engineering (CASE) Environment.* IEEE Transactions on Software Engineering, 1994. **20**(3): p. 169 - 187.

28. Daneva, M. *Measuring Reuse of SAP Requirements: a Model-based Approach.* in *Symposium on Software Reusability.* 1999. Los Angeles CA: ACM Press.

29. Lowry, G.R., *et al. A 4th Generation Information Systems Development Research Framework.* in *5th Australasian Conference on Information Systems.* 1994. Melbourne Australia: Department of Information Systems, Monash University, Australia.

30. Constantine, L. and E. Yourdon, *Structured Design.* 1975, New Jersey USA: Yourdon Press-Prentice Hall.

31. DeMarco, T., *Structured Analysis and Software Specification.* 1979, Englewood Cliffs: Yourdon Press-Prentice Hall.

32. Codd, E.F., *A Relational Model of Data for Large Shared Data Banks.* Communications of the ACM, 1970. **13**(6): p. ??

33. Chen, P., *The Entity-Relationship Model: Toward a Unified View of Data.* ACM Transactions on Database Systems, 1976. **1**(1): p. ??

34. Booch, G., *Object-Oriented Development.* IEEE Transactions on Software Engineering, 1986. **12**(2): p. 211-221.

35. Coad, P. and E. Yourdon, *Object-Oriented Analysis.* 2nd? ed. 1991, New Jersey USA: Yourdon Press-Prentice Hall.

36. Lowry, G.R. and E.E. Doroshenko. *Object-orientation in Software Engineering Education.* in *International Conference on Software Engineering: Education & Practice.* 1996. Dunedin, New Zealand: IEEE Computer Society Press.

37. Abbot, R.J., *Program Design by Informal English Descriptions.* Communications of the ACM, 1983. **26**(11): p. 882-894.

38. Guttag, J., *Abstract Data Types and the Development of Data Structures.* Communications of the ACM, 1977. **20**(6): p. 396-404.

39. Booch, G., *Object-Oriented Design with Applications.* 1st ed. 1991, Redwood City: Benjamin/Cummings.

40. Rumbaugh, J., *et al.*, *Object-oriented modeling and design.* 1991, Englewood Cliffs New Jersey: Prentice-Hall.

41. Meyer, B., *Object-Oriented Software Construction.* 1988, Englewood Cliffs New Jersey USA: Prentice-Hall.

42. Shlaer, S. and S.J. Mellor, *Object-Oriented Systems Analysis: Modelling the World in Data.* 1988, Englewood Cliffs, New Jersey: Prentice-Hall.

43. Wirfs-Brock, R. and B. Wilkerson. *Object-oriented Design: A Resposibility Driven Approach.* in *Conference on Object-oriented Programming Systems, Languages, and Applications.* 1989. New Orleans, Louisiana: ACM Press.

44. Wirfs-Brock, R.J., B. Wilkerson, and L. Weirner, *Designing Object-Oriented Software.* 1990, Englewood Cliffs New Jersey: Prentice-Hall. 341.

45. Jacobson, I., *et al.*, *Object-Oriented Software Engineering: A use case Driven Approach.* 1992, Workingham UK: ACM Press-Addison Wesley. 524.

46. Coad, P. and E. Yourdon, *Object-Oriented Design.* 1991, New Jersey USA: Yourdon Press-Prentice Hall.

47. De Champeaux, D., D. Lea, and P. Faure, *Object-Oriented Systems Development.* 1993, Reading Mass.: Addison-Wesley.

48. Humphrey, W.S., *A Discipline for Software Engineering.* 1995, Reading MA: Addison-Wesley. 789.

49. Atkinson, M., *et al. The Object-Oriented Database System Manifesto.* in *International Conference on Deductive and Object-Oriented Databases.* 1989. Kyoto, Japan.

50. Korson, T. and J.D. McGregor, *Understanding Object-Oriented: A Unifying Paradigm.* Communications of the ACM, 1990. **33**(9): p. 40-60.

51. Wirfs-Brock, R.J. and R.E. Johnson, *Surveying Current Research in Object-Oriented Design.* Communications of the ACM, 1990. **33**(9): p. 104-124.

52. Martin, B.E., C.H. Pedersen, and J. Bedford-Roberts, *An Object-Based Taxonomy for Distributed Computing Systems.* IEEE Computer, 1991. **24**(8): p. 17 - 27.

53. Bertino, E. and L. Martino, *Object-Oriented Database Management Systems: Concepts and Issues.* IEEE Computer, 1991. **24**(4): p. 33 - 47.

54. Joseph, J.V., *et al.*, *Object-Oriented Databases: Design and Implementation.* Proceedings of the IEEE, 1991. **79**(1).

55. Monarchi, D.E. and G.I. Puhr, *A Research Typology for Object-Oriented Analysis and Design.* Communications of the ACM, 1992. **35**(9): p. 35-47.

56. De Champeaux, D. and P. Faure, *A comparitive study of object-oriented analysis methods.* Journal of Object-Oriented Programming, 1992(March/April): p. 21 - 32.

57. Wegner, P., *Dimensions of Object-Oriented Modeling.* IEEE Computer, 1992. **25**(10): p. 12 - 20.

58. Fernandes, A.A., *et al.*, *Approaches to Deductive Object-Oriented Databases.* Information and Software Technology, 1992. **34**(12): p. 787 - 803.

59. English, L.P., *Object Databases at Work.* DBMS, 1992: p. 44-58.

60. Bihari, T.E. and P. Gopinath, *Object-Oriented Real-Time Systems: Concepts and Examples.* IEEE Computer, 1992. **25**(12): p. 25 - 32.

61. Fichman, R.G. and C.F. Kemerer, *Object-Oriented and Conventional Analysis and Design Methodologies: Comparison and Critique.* IEEE Computer, 1992. **25**(10): p. 22 - 39.

62. Hurson, A.R., S.H. Pakzad, and J.-b. Cheng, *Object-Oriented Database Management Systems: Evolution and Performance Issues.* IEEE Computer, 1993. **26**(2): p. 48 - 60.

63. Snyder, A., *The Essence of Objects: Concepts and Terms.* IEEE Software, 1993. **10**(1): p. 31 - 42.

64. Cerpa, N. and R. Dean, *A Review of Object-Oriented Database Concepts and thier Implementation.* Australian Journal of Information Systems, 1993.

65. Henderson, R. and B. Zorn, *A Comparison of Object-oriented Programming in Four Modern Languages.* Software Practice and Experience, 1994. **24**(11): p. 1077 - 1095.

66. Doroshenko, E.E., *Common Concepts in Object-oriented Methodologies*, in *Applied Computing & Mathematics.* 1994, University of Tasmania: Launceston. p. 179.

67. Losavio, F., A. Matteo, and F. Schlienger, *Object-oriented methodologies of Coad and Yourdon and Booch: comparison of graphical notations.* Information and Software Technology, 1994. **36**(8): p. 503-514.

68. Iivari, J., *Object-orientation as structural, functional and behavioural modelling: a comparison of six methods for object-oriented analysis.* Information and Software Technology, 1995. **37**(3): p. 155-163.

69. Guttman, M. and J.R. Matthews, *The Object Technology Revolution.* 1995, New York: Wiley & Sons. 190.

70. OMG, O.M.G., *The Common Object Request Broker: Architecture and Specification, Revision 2.0.* 1995, Farimingham, MA: Object Management Group. 100.

71. OMG, O.M.G., *CORBAServices: Common Object Services Specification.* 1996: OMG. 400.

72. OMG, O.M.G., *Object Analysis & Design Facility RFP-1.* 1996, Farmingham MA: Object Management Group. 27.

73. OMG, O.M.G., *Common Facilites RFP-5: Meta-Object Facility.* 1996, Farmingham MA: Object Management Group. 27.

74. Rational, S.C.e.a., *UML Notation Guide 1.1.* 1997, Santa Clara CA: Rational Software Corporation. 142.

75. Rational, S.C.e.a., *UML Object Constraint Specificiation Language 1.1.* 1997, Santa Clara CA: Rational Software Corporation. 32.

76. Rational, S.C.e.a., *UML Summary 1.1.* 1997, Santa Clara CA: Rational Software Corporation. 19.

77. Rational, S.C., *UML Extension for Business Modeling 1.1.* 1997, Santa Clara CA: Rational Software Corporation. 5.

78. Rational, S.C., *UML Extension for Objectory Process for Software Engineering 1.1.* 1997, Santa Clara CA: Rational Software Corporation. 5.

79. Rational, S.C.e.a., *UML Semantics 1.1.* 1997: Rational Software Corporation. 162.

80. Firesmith, D.G., *Object-oriented Requirements Analysis and Logical Design.* 1993, New York: Wiley & Sons. 575.

81. Henderson-Sellers, B., *A Book of Object-Oriented Knowledge.* 1992, New York USA: Prentice-Hall.

82. Martin, J. and J.J. Odell, *Object-Oriented Analysis and Design.* 1992, New Jersey USA: Prentice-Hall.

83. Iivari, J. *Object-oriented information systems analysis: A comparison of six object-oriented analysis methods.* in *Proceedings of the IFIP WG8.1 Working Conference on Methods and Associated Tools for the Information Systems Life Cycle.* 1994. Maastricht, The Netherlands: Elsevier.

84. Wegner, P. *Dimensions of Object-Based Language Design.* in *Conference on Object-oriented Programming Systems, Languages, and Applications.* 1987: ACM Press.

85. Lindsjorn, Y. and D. Sjoberg. *Database Concepts Discussed in an Object Oriented Perspective.* in *European Conference on Object-Oriented Programming.* 1988. Oslo, Norway: Springer-Verlag.

86. Hutt, A.T.F., *Object Analysis and Design: Description of Methods.* 1994, New York: John Wiley & Sons. 202.

87. Maciaszek, L.A., *et al. Generalization versus Aggregation in Object Application Development - the "AD-HOC" Approach.* in *Australasian Conference on Information Systems.* 1996. Hobart, Australia: Australian Computer Society Inc.

88. Graham, I. *Some Problems with Use Cases... and How to Avoid Them.* in *International Conference on Object-Oriented Information Systems.* 1996. London, UK: Springer-Verlag.

89. Jentzsch, R. *An Object-Oriented Form Driven Design Approach.* in *Australasian Conference on Information Systems.* 1993. Brisbane, Australia: Department of Commerce, University of Queensland.

90. Holmberg, S.C., *LIVING SYSTEMS APPLICATIONS: A New paradigm for Software Engineering.* Behavioral Science, 1993. **38**: p. 293 - 300.

91. Perritt, C.E. *Systems Oriented Analysis and Design Directions: A Suggested Evolution from the Object Model.* in *International Conference on Object-Oriented Information Systems.* 1997. Brinsbane, Sydney: Springer-Verlag.

92. Ram, J.D., *et al. CO-IP: An Object Model for OIS.* in *International Conference on Object-Oriented Information Systems.* 1996. London, UK: Springer-Verlag.

93. Doroshenko, E.E. *Toward a language-graphic model for CASE tool construction: helping the corner store OT vendor.* in *Australasian Conference on Information Systems.* 1996. Hobart, Australia: Australian Computer Society Inc.

94. Maier, M.W., *Integrated Modeling: A Unified Approach to System Engineering.* Journal of Systems and Software, 1996. **32**: p. 101 - 119.

95. Younessi, H., R. Smith, and D. Grant. *Towards a Systematic Approach to Object-Oriented Analysis.* in *Australasian Conference on Information Systems.* 1995. Curtin University, Australia: Australian Computer Society Inc.

96. Ledington, J. and P.W.J. Ledington. *Beyond Functional Desomposition in Soft Systems Methodology: The Decision-Variable Partitioning Approach.* in *Australasian Conference on Information Systems.* 1996. Hobart, Australia: Australian Computer Society Inc.

97. Milton, S. and C. Keen. *Linguistic-based Information Systems Modelling.* in *Australasian Conference on Information Systems.* 1996. Hobart, Australia: Australian Computer Society Inc.

98. Moody, D.L. and A. Osianlis. *Bringing Data Models To "Life": An Interactive Tool For Representing Entity Relationship Models.* in *Australasian Conference on Information Systems.* 1996. Hobart, Australia: Australian Computer Society Inc.

99. Cecez-Kecmanovic, D., I. Hawryszkiewycz, and E. Baker. *The Role Concept in Information Systems.* in *Australasian Conference on Information Systems.* 1996. Hobart, Australia: Australian Computer Society Inc.

100. Wirtz, G., *Graph-based software construction for parallel message-passing programs.* Information and Software Technology, 1994. **36**(7): p. 405 - 411.

101. Manson, G.A., S. Sahib, and C. Elamvazuthi, *Design and code derivation in the PCSC methodology.* Information and Software Technology, 1994. **36**(7): p. 413 - 417.

102. Aue, A. and M. Breu, *Distributed Information Systems: An Advanced Methodology.* IEEE Transactions on Software Engineering, 1994. **20**(8): p. 594 - 605.

103. Abbott, B., *et al.*, *Model-Based Software Synthesis.* IEEE Software, 1993. **10**(4): p. 42 - 52.

104. Bloom, T. and S.B. Zdonik. *Issues in the Design of Object-Oriented Database Programming Languages.* in *Conference on Object-oriented Programming Systems, Languages, and Applications.* 1987. Orlando, Florida: ACM Press.

105. Laenens, E. and D. Vermeir. *An Overview of OOPS+, An Object-Oriented Database Programming Language.* in *European Conference on Object-Oriented Programming.* 1988. Oslo, Norway: Springer-Verlag.

106. Lee, J.H.M. and P.K.C. Pun. *An Overview of the OLI Multiparadigm Programming Language and Its Semantics.* in *International Conference on Object-Oriented Information Systems.* 1996. Brisbane, Australia: Springer-Verlag.

107. Lee, P.J., D.J. Chen, and C.G. Chung, *An object-oriented modelling approach to system software design.* Information and Software Technology, 1994. **36**(11): p. 683 - 694.

108. Harrison, W. and H. Ossher. *Subject-Oriented Programming (A Critique of Pure Objects).* in *Conference on Object-oriented Programming Systems, Languages, and Applications.* 1993: ACM Press.

109. Naja, H. and N. Mouaddib. *Viewpoints in object-oriented databases.* in *International Conference on Object-Oriented Information Systems.* 1997. Brisbane, Australia: Springer-Verlag.

110. Nuseibeh, B., J. Kramer, and A. Finkelstein, *A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification.* IEEE Transactions on Software Engineering, 1994. **20**(10): p. 760 - 773.

111. Delannoy, X., A. Simonet, and M. Simonet. *Database Views with Dynamic Assertions.* in *International Conference on Object-Oriented Information Systems.* 1996. London, UK: Springer-Verlag.

112. Emmerich, W. *CORBA and OODBMSs in Viewpoint Development Architectures.* in *International Conference on Object-Oriented Information Systems.* 1997. Brisbane, Australia: Springer-Verlag.

113. Shilling, J.J. and P.F. Sweeney. *Three Steps to Views: Extending the Object-Oriented paradigm.* in *Conference on Object-oriented Programming Systems, Languages, and Applications.* 1989. New Orleans, Louisiana: ACM Press.

114. Kristensen, B.B. *Subject Composition by Roles.* in *International Conference on Object-Oriented Information Systems.* 1997. Brisbane, Australia: Springer-Verlag.

115. Ishikawa, Y., H. Tokuda, and C.W. Mercer, *An Object-Oriented Real-Time Programming Language.* IEEE Computer, 1992. **25**(10): p. 73.

116. Akist, M., L. Bergmans, and S. Vural. *An Object-Oriented Language-database Integration Model: The Composition Filters Approach.* in *European Conference on Object-Oriented Programming.* 1992. Utrecht, The Netherlands: Springer-Verlag.

117. Alabiso, B. *Transformation of Data Flow Analysis Models to Object-Oriented Design.* in *Conference on Object-oriented Programming Systems, Languages, and Applications.* 1988: ACM Press.

118. Briggs, T.L. and J. Werth. *A Specification Language for Object-Oriented Analysis and Design.* in *European Conference on Object-Oriented Programming.* 1994. Bologna, Italy: Springer-Verlag.

119. Shah, A.V., *et al. DSM: An Object-Relationship Modeling Language.* in *Conference on Object-oriented Programming Systems, Languages, and Applications.* 1989. New Orleans, Louisiana: ACM Press.

120. Ellison, K.S., *Developing Real-Time Embedded Software in a Market Driven Company.* 1994, New York: Wiley & Sons. 352.

121. Rational, S.C., *UML Summary 1.0.1.* 1997, Santa Clara CA: Rational Software Corporation. 15.

122. Henderson-Sellers, B. and A. Bulthuis, *Object-Oriented Metamethods.* 1997, New York, NY: Springer-Verlag. 158.

123. Kuhn, T.S., *The Structure of Scientific Revolutions.* 2nd ed. 1962, Chicago: University of Chicago Press. 210.

124. Moon, S.-H. and B.-H. Hong. *Design and Implementation of Object-Oriented Spatial Views.* in *International Conference on Object-Oriented Information Systems.* 1997. Brisbane, Australia: Springer-Verlag.

125. Herbst, H., *Business Rules in Systems Analysis: A Meta-model and Repository System.* Information Systems, 1996. **21**(2): p. 147 - 166.

126. Herbst, H., G. Knolmayer, and S. M. *The specification of business rules: a comparison of selected methodologies.* in *Proceedings of the IFIP WG8.1 Working Conference on Methods and Associated Tools for the Information Systems Life Cycle.* 1994. Maastricht, The Netherlands: Elsevier.

127. Vranes, S. and M. Stanojevic, *Integrating Multiple Paradigms within the Blackboard Framework.* IEEE Transactions on Software Engineering, 1995. **21**(3): p. 244 - 262.

128. Peterson, J.L., *Petri Nets.* ACM Computing Surveys, 1977. **9**(3): p. 223 - 252.

129. Fowler, M. and K. Scott, *UML Distilled.* 1997, Reading, MA: Addison-Wesley. 179.

130. Yau, S.S. and M.U. Caglayan, *Distributed Software Design Representation Using Modified Petri Nets.* IEEE Transactions on Software Engineering, 1983. **9**(6): p. 733 - 745.

131. Papelis, Y.E. and T.L. Casavant, *Specification and Analysis of Parallel/Distributed Software and Systems by Petri Nets With Transition Enabling Functions.* IEEE Transactions on Software Engineering, 1992. **18**(3): p. 252 - 261.

132. Rahayu, W., *et al. Aggregation versus Association in Object Modelling and Databases.* in *Australasian Conference on Information Systems.* 1996. Hobart, Australia: Australian Computer Society Inc.

133. Jacobson, I., M. Griss, and P. Jonsson, *Software Reuse: Architecture, Process, and Organization for Business Success.* 1997, New York NY: ACM Press-Addison Wesley. 499.

134. Pressman, R.S., *Software Engineering: A Practitioner's Approach.* 5th Ed ed. 2001, Boston IL: McGraw-Hill. 860.

135. Cusumano, M.A., *Japan's Software Factories.* 1991, New York, NY: Oxford. 513.

136. Pfleeger, S.L., *Software Engineering: Theory and Practice.* 1998, Upper Saddle River, NJ: Prentice-Hall. 568.

137. Wasserman, A.I., *Toward a Discipline of Software Engineering.* IEEE Software, 1996. **13**(6): p. 23 - 31.

138. Wastell, D.G., *The fetish of technique: methodology as a social defence.* Information Systems Journal, 1996(6): p. 25 - 40.

139. Senn, J.A., *Analysis and Design of Information Systems.* 2nd ed. Computer Science. 1989, New York: McGraw-Hill. 853.

140. Oivo, M. and V.R. Basili, *Representing Software Engineering Models: The TAME Goal Oriented Approach.* IEEE Transactions on Software Engineering, 1992. **18**(10): p. 886 - 898.

141. Basili, V.R. and H.D. Rombach, *The TAME Project: Towards Improvement-Oriented Software Environments.* IEEE Transactions on Software Engineering, 1988. **14**(6): p. 759 - 773.

142. DeMarco, T., *Controlling Software Projects: Management, Measurement & Estimation.* 1982, Englewood Cliffs, NJ: PTR Prentice-Hall. 284.

143. Jones, C., *Applied Software Measurement: Assuring Productivity and Quality.* Software Engineering. 1991, New York: McGraw-Hill. 493.

144. Verner, J. and G. Tate, *A Software Size Model.* IEEE Transactions on Software Engineering, 1992. **18**(4): p. 265 - 278.

145. Roche, J. and M. Jackson, *Software measurement methods: recipies for success?* Information and Software Technology, 1994. **36**(3): p. 173 - 189.

146. Tervonen, I., *Support for Quality-Based Design and Inspection.* IEEE Software, 1996. **13**(1): p. 44 - 54.

147. Offen, R.J. and R. Jeffery, *Establishing Software Measurement Programs.* IEEE Software, 1997. **14**(2): p. 45 - 53.

148. Pfleeger, S.L., *Maturity, Models, and Goals: How to Build a Metrics Plan.* Journal of Systems and Software, 1995. **31**(2): p. 143 - 155.

149. Humphrey, W.S., *Managing the Software Process.* 1989, Reading Massachusetts USA: Addison-Wesley. 494.

150. Rolland, C., C. Souveyet, and M. Moreno, *An Approach for Defining Ways-Of-Working.* Information Systems, 1995. **20**(4): p. 337 - 359.

151. Pfleeger, S.L. and C. McGowan, *Software Metrics in the Process Maturity Framework.* Journal of Systems and Software, 1990. **12**: p. 255 - 261.

152. ter Hofstede, A.H.M. and T.F. Verhoef, *On The Fiesibility of Situational Method Engineering.* Information Systems, 1997. **22**(6/7): p. 401 - 422.

153. Vlasbolm, G., D. Rijsenbrij, and M. Glastra, *Flexibalization of the methodology of system development.* Information and Software Technology, 1995. **37**(11): p. 595 - 607.

154. Van Slooten, K. and S. Brinkkemper. *A Method Engineering Approach to Information Systems Development.* in *IFIP WG8.1 Working Conference on Information System Development Process.* 1993. Como, Italy: Elsevier.

155. Brinkkemper, S., *Formalisation of Information Systems Modelling,* . 1990, University of Nijmegen.

156. Brinkkemper, S., *Method Engineering: Engineering of information systems development methods and tools.* Information and Software Technology, 1996. **38**(4): p. 275 - 280.

157. Falkenberg, E.D., *et al.*, *A Framework of Information System Concepts,* . 1998, Department of Computer Science, University of Leiden: Leiden.

158. Iivari, J., *Hierachical spiral model for information system and software development: Part 1: theoretical background.* Information and Software Technology, 1990. **32**(6): p. 386 - 396.

159. Iivari, J. *Object-oriented information systems analysis: A framework for object identification.* in *Hawaii International Conference on System Sciences.* 1991. Hawaii: IEEE Computer Society Press.

160. Boehm, B.W., *A Spiral Model of Software Development and Enhancement.* IEEE Computer, 1988. **21**(5): p. 61 - 72.

161. Nuseibeh, B., A. Finkelstein, and J. Kramer, *Method engineering for multi-perspecive software development.* Information and Software Technology, 1996. **38**(4): p. 267 - 274.

162. Van Slooten, K., *Situated Method Engineering.* Information resources management journal, 1996. **9**(3): p. 24 - 31.

163. ter Hofstede, A.H.M. and T.F. Verhoef, *Meta-CASE: Is the game worth the candle?* Information Systems Journal, 1996(6): p. 41 - 68.

164. Brodman, J.G. and D.L. Johnson. *What Small Businesses and Small Organizations Say About the CMM.* in *International Conference on Software Engineering.* 1994. Sorrenta, Italy: IEEE Computer Society Press.

165. Harmsen, F. and S. Brinkkemper. *A Language and Tool for the Engineering of Situational Methods for Information Systems Development.* in *ISD.* 1994. Bled, Slovenia.

166. Saeki, M., *A Meta-Model for Method Integration.* Information and Software Technology, 1998. **39**(14-15): p. 925 - 932.

167. Krasner, H., *et al., Lessons Learned from a Software Process Modelling System.* Communications of the ACM, 1992. **35**(9): p. 91-100.

168. Curtis, B., M.I. Kellner, and J. Over, *Process Modelling.* Communications of the ACM, 1992. **35**(9): p. 75 - 90.

169. Belkhatir, N. and M. Ahmed-Nacer, *Major Issues on Process Software Engineering Environments.* Information Sciences, 1995. **83**: p. 1 - 21.

170. Belkhatir, N. and W.L. Melo, *Towards an Integration of Software Product and Software Process Modeling.* Integrated Computer Aided Engineering, 1996. **3**(1): p. 36 - 50.

171. Kelly, S., *A matrix editor for a metaCASE environment.* Information and Software Technology, 1994. **36**(6): p. 361 - 371.

172. Joosten, S. *Lazy Functional Meta-CASE Programming.* in *Proceedings of the IFIP TC8, WG8.1/8.2 Working Conference on Method Engineering.* 1996. Atlanta, USA: Chapman & Hall.

173. Mi, P. and W. Scacchi, *A meta-model for formulating knoledge-based models of software development.* Decision Support Systems, 1996. **17**: p. 313 - 330.

174. McDonald, C. *Information Systems Modelling and Conceptual Graphs.* in *Australasian Conference on Inforamtion Systems.* 1996. Hobart, Australia: Australian Computer Society Inc.

175. Odell, J.J., *Introduction to Method Engineering.* Object Magazine, 1995. **5**(5): p. 69 - 71, 91.

176. Odell, J.J. *Keynote paper: a primer to method engineering.* in *IFIP TC 8, WG8.1/8.2 Working Conference on Method Engineering.* 1996. Atlanta USA: Chapman & Hall.

177. Akist, M. and L. Bergmans. *Obstacles in Object-Oriented Software Development.* in *Conference on Object-oriented Programming Systems, Languages, and Applications.* 1992: ACM Press.

178. Henderson-Sellers, B. and D.G. Firesmith, *COMMA: Proposed core model.* Journal of Object-Oriented Programming, 1997. **9**(8).

179. Henderson-Sellers, B., I. Graham, and D.G. Firesmith, *Methods unification: The OPEN methodology.* Journal of Object-Oriented Programming, 1997. **10**(2): p. 41 - 43, 55.

180. Henderson-Sellers, B., *OPEN: Toward method convergence?* IEEE Computer, 1996. **29**(4): p. 86 - 89.

181. Jacobson, I., *Is Object Technology Software's Industrial Platform?* IEEE Software, 1993. **10**(1): p. 24 - 30.

182. Fayad, M.E., *et al.*, *Using the Shlaer-Mellor Object-Oriented Analysis Method.* IEEE Software, 1993. **10**(2): p. 43 - 52.

183. Wynekoop, J.L. and N.N. Russo, *Studying system development methodologies: an examination of research methods.* Information Systems Journal, 1997. **7**(1): p. 47 - 65.

184. Lubars, M., C. Potts, and C. Richter. *A review of the state of the art in requirements modeling.* in *International Symposium on Requirements Engineering.* 1992: IEEE Computer Society Press.

185. Bryant, T. and A. Evans, *OO oversold : Those objects of obscure desire.* Information and Software Technology, 1994. **36**(1): p. 35-42.

186. Booch, G., *Object Solutions: Managing the Object-Oriented Project.* 1996, Melno Park, CA: Addison-Wesley. 323.

187. Ou, Y. *On Using UML Class Diagram for Object-oriented Database Design - Specification of Integrity Constraints.* in *UML International Workshop.* 1998. Mulhouse, France: ESSAIM.

188. Schach, S.R., *Classical and Object-Oriented Software Engineering with UML and C++.* 4 ed. 1999, Boston MA: McGraw-Hill. 616.

189. Monarchi, D.E.M. *Methodology Standards: Help or Hinderance? (PANEL).* in *Conference on Object-oriented Programming Systems, Languages, and Applications.* 1994. Portland, Oregon: ACM Press.

190. Coleman, D.M. *(PANEL) UML: the language of blueprints for software?* in *Conference on Object-oriented Programming Systems, Languages, and Applications.* 1997. Atlanta, Georgia: ACM Press.

191. Mehandjiska-Stavreva, D., D. Page, and J. Ham. *Template Generator for a Methodology Independent Object-Oriented CASE Tool.* in *International Conference on Object-Oriented Information Systems.* 1995. Dublin Ireland: Springer-Verlag.

192. Oei, H. and E. Falkenberg. *Harmonisation of information system modelling and specification techniques.* in *Proceedings of the IFIP WG8.1 Working Conference on Methods and Associated Tools for the Information Systems Life Cycle.* 1994. Maastricht, The Netherlands: Elsevier.

193. Nissen, H.W., *et al.*, *Managing Multiple Requirements Perspectives with Metamodels.* IEEE Software, 1996. **13**(2): p. 37 - 47.

194. Fayad, M.E., *et al.*, *Adapting an Object-Oriented Development Method.* IEEE Software, 1994. **11**(3): p. 68 - 76.

195. Nguyen, G.T., D. Rieu, and J. Escamilla. *An Object Model for Engineering Design.* in *European Conference on Object-Oriented Programming.* 1992. Utrecht, The Netherlands: Springer-Verlag.

196. Fitsillis, P., *Object-oriented development for telecommunication services.* Information and Software Technology, 1995. **37**(1): p. 15 - 22.

197. Kleppe, A., J. Warmer, and S. Cook. *Infomal formality? The Object Constraint Language ans its application in the UML metamodel.* in *UML'98 International Workshop.* 1998. Mulhouse, France: ESSAIM.

198. Jacobson, I., G. Booch, and J. Rumbaugh, *The Unified Software Development Process.* 1999, Reading MA: Addison-Wesley. 463.

199. Mancl, D.M. *(PANEL) Tailoring OO analysis and design methods.* in *Conference on Object-oriented Programming Systems, Languages, and Applications.* 1995. Austin, Texas: ACM Press.

200. Jacobson, I., *Time for a cease-fire in the methods war.* Journal of Object-Oriented Programming, 1993. **6**(4): p. 6, 84.

201. Mellor, S.J., *et al.*, *Premature methods standardization considered harmful.* Journal of Object-Oriented Programming, 1993. **6**(4): p. 8, 85.

202. Yourdon, E., *Object-Oriented Systems Design: An Integrated Approach.* 1994, Englewood Cliffs, NJ: Prentice-Hall. 400.

203. Warmer, J. and A. Kleppe, *The Object Constraint Language*. 1999, Reading MA: Addison-Wesley. 144.

204. Beringer, D. *Limits of Seamlessness in Object-oriented Software Development*. in *TOOLS Europe*. 1994: Prentice-Hall.

205. Fernstrom, C., K.-H. Narfelt, and L. Ohlsson, *Software Factory Principles, Architecture, and Experiments.* IEEE Software, 1992. **9**(2): p. 36 - 44.

206. Fuggetta, A., *A Classification of CASE Technology.* IEEE Computer, 1993. **26**(12): p. 25 - 38.

207. Korson, T.D. and V.K. Vaishnavi, *Managing Emerging Software Technologies: A Technology Transfer Framework.* Communications of the ACM, 1992. **35**(9): p. 101 - 111.

208. McChesney, I.R., *Toward a classification scheme for software process modelling approaches.* Information and Software Technology, 1995. **37**(7): p. 363 - 374.

209. Doroshenko, E.E. *A Model for Research Surveying in Software Methodologies*. in *Australasian Conference on Information Systems*. 1997. Adelaide, Australia: Australian Computer Society Inc.

210. Whiddett, R.J. and M.A. Bailey. *Complexity and Maintinence: A Comparitive Study of Object-Oriented and Structured Methodologies*. in *International Conference on Object-Oriented Information Systems*. 1997. Brisbane, Australia: Springer-Verlag.

211. Leach, R., *Methods of Measuring Software Reuse for the Prediction of Maintenance Effort.* Journal of Software Maintenance: Research and Practice, 1996. **8**(5): p. 309 - 320.

212. Chen, Y.-F., B. Krishnamurty, and K.-P. Vo, *An Objective Reuse Metric: Model and Methodology*, in *Proceedings of the Fifth European Software Engineering Conference*. 1995, Springer-Verlag: Berlin, Germany. p. 109 - 123.

213. Melo, W.L., L.C. Briand, and V.R. Basili, *Measuring the Impact of Reuse on Quality and Productivity in Object-Oriented Systems*, . 1995, Dept. of Computer Science, University of Maryland: College PArk.

214. Basili, V.R., L.C. Briand, and W.L. Melo, *How Reuse Influences Productivity in Object-Oriented Systems.* Communications of the ACM, 1996. **39**(10): p. 104 - 116.

215. Bieman, J.M. and S. Karunanithi, *Measurement of Language-Supported Reuse in Object-Oriented and Object-Based Software.* Journal of Systems and Software, 1995. **30**(3): p. 271 - 293.

216. Fewster, R. and E. Mendes. *Measurement, Prediction and Risk Analysis for Web Applications*. in *International Symposium on Software Metrics*. 2001: IEEE Computer Society Press.

217. Morasca, S. and G. Ruhe, *A hybrid approach to analyse empirical software engineering data and its application to predict module fault-proneness in maintenance.* Journal of Systems and Software, 2000. **53**: p. 225 - 237.

218. Li, W., *An Empirical Study of Software Reuse in Reconstructive Maintenance.* Journal of Software Maintenance: Research and Practice, 1997. **9**: p. 69 - 83.

219. Chen, J.Y. and J.F. Lu, *A new metric for object-oriented design.* Information and Software Technology, 1993. **35**(4): p. 232 - 240.

220. Selby, R.W., *Quantitative Studies of Software Reuse*, in *Software Reusability: Applications and Experience*, T.J. Biggerstaff and A.J. Perlis, Editors. 1989, ACM Press-Addison Wesley: Reading MA. p. 213 - 233.

221. Lewis, J.A., *et al. An Empirical Study of the Object-Oriented Paradigm and Software Reuse*. in *Conference on Object-oriented Programming Systems, Languages, and Applications*. 1991: ACM Press.

222. Subramanian, G. and W. Corbin, *An empirical study of certain object-oriented software metrics.* Journal of Systems and Software, 2001. **59**: p. 57 - 63.

223. Succi, G., L. Benedicenti, and T. Vernazza, *Analysis of the Effects of Software Reuse on Customer Satisfaction in an RPG Environment.* IEEE Transactions on Software Engineering, 2001. **27**(5): p. 473 - 479.

224. Curry, W., *et al. Empirical Analysis of the Correlation between Amount-of-Reuse Metrics in the C Programming Language*. in *Symposium on Software Reusability*. 1999. Los Angeles CA: ACM Press.

225. Bieman, J.M., *Deriving Measures of Software Reuse in Object-Oriented Systems*, in *BCS-FACS Workshop on Formal Aspects of Measurement*. 1991, Springer-Verlag: London, UK. p. 63 - 83.

226. Hogen, J., *An Analysis of OO Software Metrics*, . 1997, Department of Computer Science, University of Warwick: Coventry.

227. Lim, W.C. *Reuse Economics: A Comparison of Seventeen Models and Directions for Future Research.* in *International Conference on Software Reuse*. 1996: IIEEE Computer Society Press.

228. Hitz, M. *Measuring Reuse Attributes In Object-Oriented Systems.* in *International Conference on Object-Oriented Information Systems.* 1995. Dublin, Ireland: Springer-Verlag.

229. Poulin, J.S., J.M. Caruso, and D.R. Hancock, *The business case for software reuse.* IBM Systems Journal, 1993. **32**(4): p. 567 - 594.

230. Devanbu, P., *et al. Analytical and Emperical Evaluation of Software Reuse Metrics.* in *International Conference on Software Engineering.* 1996. Berlin, Germany: IEEE Computer Society Press.

231. Devanbu, P., *et al.*, *Analytical and Emperical Evaluation of Software Reuse Metrics*, . 1996, Dept. of Computer Science: College Park.

232. Bosua, R. and S. Brinkkemper. *A Structured Approach for Integration of Object-oriented and Conventional CASE tools.* in *International Conference on Object-Oriented Information Systems.* 1995. Dublin, Ireland: Springer-Verlag.

233. Kelly, S. and K. Smolander, *Evolution and issues in metaCASE.* Information and Software Technology, 1996. **38**(4): p. 261 - 266.

234. Harmsen, F., S. Brinkkemper, and H. Oei. *Situational Method Engineering for Information System Project Approaches.* in *Proceedings of the IFIP WG8.1 Working Conference on Methods and Associated Tools for the Information Systems Life Cycle.* 1994. Maastricht, The Netherlands: Elsevier.

235. Glasson, B.C., *Model of system evolution.* Information and Software Technology, 1989. **31**(7): p. 351 - 361.

236. Iivari, J., *Hierachical spiral model for information system and software development: Part 2: design process.* Information and Software Technology, 1990. **32**(7): p. 450 - 460.

237. Saeki, M., *et al. A Meta-Model for Representing Software Specification & Design Methods.* in *IFIP WG8.1 Working Conference on Information System Development Process.* 1993. Como, Italy: Elsevier.

238. Bailey, J.W. and V.R. Basili. *A Meta-Model for Software Development Resource Expenditures.* in *International Conference on Software Engineering.* 1981. San Diego CA: IEEE Computer Society Press.

239. Rossi, M. and S. Brinkkemper, *Metrics in Method Engineering*, in *Lecture Notes in Computer Science.* 1995, Springer-Verlag: Berlin, Germany. p. 200 - 216.

240. Bubenko, J., *et al. Facilitating "Fuzzy to Formal" Reqiurements Modelling.* in *1st International Conference on Requirements Engineering.* 1994. Colorado Springs, Colorado: IEEE Computer Society Press.

241. Czejdo, B. and M.C. Taylor, *Integration of Information Systems Using an Object-Oriented Approach.* The Computer Journal, 1992. **35**(5): p. 501 - 513.

242. Pastor, E.A. and R.T. Price. *Using Metamodels of Methodologies to determine the needs for reusability support.* in *Symposium on Software Reusability.* 1997. Boston, Massachusetts: ACM Press.

243. De Antonellis, V., S. Castano, and L. Vandoni, *Bulding Reusable Components Through Project Evolution Analysis.* Information Systems, 1994. **19**(3): p. 259 - 274.

244. Castano, S. and V. De Antonellis. *A Constructive Approach to Reuse of Conceptual Components.* in *Advances in Software Reuse.* 1993: IEEE Computer Society Press.

245. Steele, P.M. and J. Han. *A Layered Architecture for Describing Information Systems Development Methodologies.* in *Australasian Conference on Information Systems.* 1996. Hobart, Australia: Australian Computer Society.

246. Dominguez, E., M.A. Zapata, and J. Rubio, *A Conceptual Approach to Meta-Modelling*, in *Advanced Information Systems Engineering: 9th International Conference.* 1997, Springer-Verlag: Berlin, Germany. p. 319 - 332.

247. Biggerstaff, T.J. and A.J. Perlis, *Software Reusability: Concepts and Models.* Frontier. Vol. 1. 1989, Reading, MA: ACM Press-Addison Wesley. 425.

248. Briand, L.C., S. Morasca, and V.R. Basili, *Property-Based Software Engineering Measurement.* IEEE Transactions on Software Engineering, 1996. **22**(1): p. 68 - 85.

249. Gustafson, D.A., J.T. Tan, and P. Weaver. *Software Measure Specification.* in *Symposium on the Foundations of Software Engineering.* 1993: ACM Press.

250. Chidamber, S., R. and C.F. Kemerer. *Towards a Metric Suite for Object-Oriented Design.* in *Conference on Object-oriented Programming Systems, Languages, and Applications.* 1991: ACM Press.

251. Chidamber, S., R. and C.F. Kemerer, *A Metrics Suite for Object-Oriented Design.* IEEE Transactions on Software Engineering, 1994. **20**(6): p. 476 - 493.

Chapter 3

1. Backus, J.W. *The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference.* in *International Conference on Information Processing.* 1959. Paris, France: Unesco.

2. Doroshenko, E.E. *A Model for Research Surveying in Software Methodologies.* in *Australasian Conference on Information Systems.* 1997. Adelaide, Australia: Australian Computer Society Inc.

3. Doroshenko, E.E. and F.A. Lear, *Refinement of Research Surveying in Software Methodologies by Analogy: Finding Your Patch.* Australian Journal of Information Systems, 1999. **6**(2): p. 13 - 35.

4. Doroshenko, E.E. *Toward a language-graphic model for CASE tool construction: helping the corner store OT vendor.* in *Australasian Conference on Information Systems.* 1996. Hobart, Australia: Australian Computer Society Inc.

5. Naur, P., *Revised report on the algorithmic language Algol.* Communications of the ACM, 1963. **6**(1): p. 1-17.

6. Sethi, R., *Programming Languages: Concepts and Constructs.* Addison-Wesley Series in Computer Science, ed. M.A. Harrison. 1989, Reading, MA: Addison-Wesley. 478.

7. Booch, G., *Object-Oriented Analysis and Design with Applications.* 2nd ed. 1994, Redwood City: Benjamin/Cummings. 589.

8. De Champeaux, D., D. Lea, and P. Faure, *Object-Oriented Systems Development.* 1993, Reading Mass.: Addison-Wesley.

9. Borland, *Borland C++ User's Guide.* 1996, Scotts Valley: Borland International Inc. 630.

10. Borland, *Borland C++ Programmer's Guide.* 1996, Scotts Valley: Borland International Inc. 769.

11. Borland, *Borland C++ ObjectWindows Programmer's Guide.* 1996, Scotts Valley: Borland International Inc. 538.

12. Borland, *Borland C++ Language Reference.* 1996, Scotts Valley: Borland International Inc. 1550.

13. Borland, *Borland C++ ObjectWindows Reference.* 1996, Scotts Valley: Borland International Inc. 1036.

Chapter 4

No Citations

Chapter 5

1. Peterson, J.L., *Petri Nets.* ACM Computing Surveys, 1977. **9**(3): p. 223 - 252.

2. Peterson, J.L., *Petri net theory and the modelling of systems.* 1981, Englewood Cliffs NJ: Prentice-Hall.

3. Yau, S.S. and M.U. Caglayan, *Distributed Software Design Representation Using Modified Petri Nets.* IEEE Transactions on Software Engineering, 1983. **9**(6): p. 733 - 745.

4. Checkland, P.B., *Systems Thinking, Systems Practice.* 1981, Chichester UK: Wiley & Sons.

5. Checkland, P.B. and J. Scholes, *Soft Systems Methodology in Action.* 1990, Chichester UK: Wiley & Sons.

6. Herbst, H., *Business Rules in Systems Analysis: A Meta-model and Repository System.* Information Systems, 1996. **21**(2): p. 147 - 166.

7. Herbst, H., G. Knolmayer, and S. M. *The specification of business rules: a comparison of selected methodologies.* in *Proceedings of the IFIP WG8.1 Working Conference on Methods and Associated Tools for the Information Systems Life Cycle.* 1994. Maastricht, The Netherlands: Elsevier.

8. Holmberg, S.C., *LIVING SYSTEMS APPLICATIONS: A New paradigm for Software Engineering.* Behavioral Science, 1993. **38**: p. 293 - 300.

9. Miller, J.G., *Living Systems.* 1978, New York NY: McGraw-Hill.

10. Ellison, K.S., *Developing Real-Time Embedded Software in a Market Driven Company.* 1994, New York: Wiley & Sons. 352.

11. Buhr, R.J.A. and R.S. Casselman. *Timethread-Role Maps for Object-Oriented Design of Real-Time and Distributed Systems.* in *Conference on Object-oriented Programming Systems, Languages, and Applications.* 1994. Portland, Oregon: ACM Press.

12. Fitsillis, P., *Object-oriented development for telecommunication services.* Information and Software Technology, 1995. **37**(1): p. 15 - 22.

13. Lee, P.J., D.J. Chen, and C.G. Chung, *An object-oriented modelling approach to system software design.* Information and Software Technology, 1994. **36**(11): p. 683 - 694.

14. Harrison, W. and H. Ossher. *Subject-Oriented Programming (A Critique of Pure Objects).* in *Conference on Object-oriented Programming Systems, Languages, and Applications.* 1993: ACM Press.

15. Shilling, J.J. and P.F. Sweeney. *Three Steps to Views: Extending the Object-Oriented paradigm.* in *Conference on Object-oriented Programming Systems, Languages, and Applications.* 1989. New Orleans, Louisiana: ACM Press.

16. Delannoy, X., A. Simonet, and M. Simonet. *Database Views with Dynamic Assertions.* in *International Conference on Object-Oriented Information Systems.* 1996. London, UK: Springer-Verlag.

17. Ram, J.D., *et al. CO-IP: An Object Model for OIS.* in *International Conference on Object-Oriented Information Systems.* 1996. London, UK: Springer-Verlag.

18. Jackson, R.B., D.W. Embley, and S.N. Woodfield, *Developing Formal Object-Oriented Requirements Specifications: A Model, Tool, and Technique.* Information Systems, 1995. **20**(4): p. 273 - 289.

19. Wirfs-Brock, R.J., B. Wilkerson, and L. Weirner, *Designing Object-Oriented Software.* 1990, Englewood Cliffs New Jersey: Prentice-Hall. 341.

20. Firesmith, D.G., *Object-oriented Requirements Analysis and Logical Design.* 1993, New York: Wiley & Sons. 575.

21. Beckstein, C., G. Gorz, and M. Tielemann, *FORK: A System for Object- and Rule-Oriented Programming,* in *European Conference on Object-Oriented Programming.* 1987, Springer-Verlag: Berlin, Germany. p. 253 - 264.

22. Lee, J.H.M. and P.K.C. Pun. *An Overview of the OLI Multiparadigm Programming Language and Its Semantics.* in *International Conference on Object-Oriented Information Systems.* 1996. Brisbane, Australia: Springer-Verlag.

Chapter 6

1. Poulin, J.S., J.M. Caruso, and D.R. Hancock, *The business case for software reuse.* IBM Systems Journal, 1993. **32**(4): p. 567 - 594.

2. Frakes, W. and C. Terry, *Software Reuse: Metrics and Models.* ACM Computing Surveys, 1996. **28**(2): p. 415 - 435.

3. Mili, H., F. Mili, and A. Mili, *Reusing Software: Issues and Research Directions.* IEEE Transactions on Software Engineering, 1995. **21**(6): p. 528 - 562.

4. Banker, R.D., *et al.*, *Automating Output Size and Reuse Metrics in a Repository-Based Computer-Aided Software Engineering (CASE) Environment.* IEEE Transactions on Software Engineering, 1994. **20**(3): p. 169 - 187.

5. Gall, J., *Systematics: How Systems Really Work and How They Fail.* 1986.

6. Rational, S.C.e.a., *UML Semantics 1.1.* 1997: Rational Software Corporation. 162.

7. Beringer, D. *Limits of Seamlessness in Object-oriented Software Development.* in *TOOLS Europe.* 1994: Prentice-Hall.

8. Pfleeger, S.L., *Software Engineering: Theory and Practice.* 1998, Upper Saddle River, NJ: Prentice-Hall. 568.

9. Lowry, G.R., *Systems Analysis: A 4th Generation Approach.* 1993, Dubuque: Wm C. Brown. 428.

10. Steele, P.M. and A.B. Zaslavsky. *CASE Tool support for Integration of System Modelling Techniques.* in *Australasian Conference on Information Systems.* 1993. Melbourne, Australia.

11. Dampney, C.N.G. and R.M. Colomb, *Semantic correspondence in integrating CASE tool repository schemas.* Information and Software Technology, 1994. **36**(2): p. 87 - 96.

12. Shaw, M. and D. Garlan, *Software Architecture: Perspecives on an Emerging Discipline.* 1996, Upper Sadddle River: Prentice-Hall.

13. Fenton, N.E., *Software Metrics: A rigorous approach.* 1991, London: Chapman & Hall. 337.

14. Whiddett, R.J. and M.A. Bailey. *Complexity and Maintinence: A Comparitive Study of Object-Oriented and Structured Methodologies.* in *International Conference on Object-Oriented Information Systems.* 1997. Brisbane, Australia: Springer-Verlag.

15. Leach, R., *Methods of Measuring Software Reuse for the Prediction of Maintenance Effort.* Journal of Software Maintenance: Research and Practice, 1996. **8**(5): p. 309 - 320.

16. Pree, W., *Design Patterns for Object-Oriented Software Development.* 1995, Reading MA: Addison-Wesley.

List of Abbreviations

aa	Amount Added
AA	Automation assessment
ag	Amount Generated
ams	Size of Application Model
ang	Amount Not Generated
anr	Amount Not Reused
AOR	Abbreviation for Amount of Reuse
ar	Amount Reused
Base DefL	Base Definition Location
Based DefM	Based Definition Method
cms	Size of Complete Model
DefL	Definition Location
DefM	Definition Method
gms	Size of Generated Model
IMea	Interpretation or Meaning
lms	Size of library Model
MetLn	Meta-Level n, n is an integer.
MLC	Meta-Level Component
MLCDep	MLC Dependency
MLCDesc	MLC Descriptor
MMA	Meta-Model Architecture
MT	Measurement testing
OO	Object-Oriented
OOM	Object-Oriented Methodologies
paam	Percentage Added in Application Model
pccm	Percentage Contribution to Complete Model
plmnr	Percentage of Library Model Not Reused
png	Percent Not Generated
pram	Percentage of Reuse in Application Model
pwg	Percentage of Waste Generation
SDM	Structured Data Methodologies

SMC	Software model classification
sms	Size of Source Model
SMTC	Software model type classification
SPM	Structured Process Methodologies
UML	Unified Modelling Language
wg	Waste Generated

Glossary

Abstraction	A process where the significant aspects of something are highlighted, ignoring any unnecessary details. See for further details.
Action	Something done or performed, when either an event has occurred or as part of a state in a state transition model.
Activity	A process that cannot be broken down into subphases or steps. Activities set specific aims in a method process. For example, Identify classes is an activity. Requirements elicitation is not an activity.
Aggregate	The name given to a class that contains a number of components or parts in an aggregation relationship. For example, a car contains an engine. The car is the aggregate.
Aggregation	A relationship between two classes where one class called the aggregate or whole contains a number of other classes called the components or parts. See for further details. For example, a car contains an engine. There is an aggregation relationship between the car and the engine.
Amount Added	The difference between the amount reused and the size of an application model in composition reuse.
Amount Generated	The contribution of a generated model to a complete model through generation reuse.
Amount Not Generated	The difference between the amount generated and the size of a complete model in generation reuse.
Amount Not Reused	The part of a library model that is not reused through composition reuse.
Amount of reuse	A measure of how much of one product or entity is reused in another entity. Amount of reuse measures can exist for internal or external, and composition or generation reuse.
Amount Reused	The contribution of a library model to an application model through composition reuse.
Analysis (Measurement Process)	A phase in the measurement process where measures collected (data) are analysed using pre-defined procedures and techniques.
Analysis process	The process used in a methodology phase to analyse data and determine if any hypotheses are supported or not supported. The analysis process is composed of analysis specifications, assessment criteria, analysis report input, analysis report output, and the analysis process description.

Analysis process description	A step by step description of the process used in a methodology phase to calculate the values for hypotheses measures, generate any analysis reports, and modify any analysis reports.
Analysis report	A report designed to document results for hypotheses measures after execution of the experiment in a methodology phase.
Analysis Report Input	The analysis reports used in a methodology phase.
Analysis report output	Analysis reports generated or modified after analysis of data.
Analysis specifications	Hypothesis measures used to quantity support for hypotheses tested in a methodology phase.
Application Model	The product that reuses a library model in composition reuse.
Approach to Reuse	An synonym for the principle of reuse
Assessed Components	The part of the measurement framework that is used to test hypotheses in a methodology phase. This is usually a list of the meta-level components.
Assessment criteria	Assessment criteria species the values required for hypothesis measures used in a methodology phase (analysis specifications) to support or not support hypotheses under investigation.
Assessment Framework	The experiments conducted to test the measurement framework against the hypotheses, that is, the experiments are designed to validate or invalidate the hypotheses.
Association	A relationship between classes based on the meaning behind some dialog in some specific context. For example, Jim manages jack, where Jim is a manager and Jack is an Employee. The association is manages.
Attribute (Measurement)	An aspect or feature of an entity that is measured. For example, size is an attribute of a class diagram.
Attribute (object-oriented modelling)	The name given to some part of an object that represents part of the state of an object. This includes data. For example, a book (the object) has a title and author (the attributes).
Automation assessment methodology phase	An experiment designed to see if the measurement framework used to measure the amount of reuse has any limitations.
Automation Element Specification	The structure of a software tool based on some aspect of the implementation. For example, menu items.

Automation Specification	The structure of a software tool that implements the measurement framework.
Automation Specification Mapping	The relationship between the measurement framework expressed in the meta-model architecture the components in a software tool that implement the measurement framework.
Axis Component	The axis component classifies dimensions of a research surveying tool.
Base Definition Location (Base DefL)	A meta-level component that is not contained by other meta-level components.
Base Definition Method (Base DefM)	A meta-level component that is not defined using other meta-level components.
Base indicator	A hypotheses measure used as part of a coverage or limitation indicator.
Class	A set of objects with a common set of attributes and operations. For example, all cars have an engine and wheels.
Class Diagram	The notation used to represent a class model.
Class Model	A software model based on the object-oriented paradigm that models the static structure of a system. A class model is an example of a software model type.
Classification	The act of grouping related things together based on their common features. For example, vehicles can be classified as cars or trucks.
Client (object-oriented methodologies)	An Object that is sender of messages to a supplier. The Client requests performance of one or more services of the supplier. Clients are connected to suppliers via a link.
Client (Tailoring of Software Methodologies)	The reason or cause of tailoring a software methodology. Three reasons are used, namely, the organisation, the project, and the domain.
Client Class	The name given to the class of an object that is a client in a communication relationship.
Collaboration Diagram	The notation used to represent a collaboration model.
Collaboration Model	A software model in the object-oriented paradigm that captures the dynamic behaviour of a set of objects over an interval of time
Collection	A phase in the measurement process where measures are collected during a software project.

Collection process	The process used in a methodology phase to collect data suitable for testing one or more hypotheses. The collection process is composed of assesses components, hypotheses tested, data specifications, data collection instruments, and the collection process description.
Collection process description	A step-by-step description of the experiment.
Communication	A relationship between classes in which one class, called the client class uses the operations of another class, called the supplier class.
Complete Model	The model that reuses the generated model in generation reuse.
Complexity Management	The techniques used to make system model views easier to understand. For example, class diagrams can use packages as a substitute for classes and relationships.
Component (object-oriented methodologies)	The class that is part of another class, called the aggregate in an aggregation relationship.
Component Specification	The structure of a software tool based on a basic architecture of components that implement a set of related functions.
Composite Class	A class that contains other classes via aggregation or composition. Composite classes can be represented in a number of ways for complexity management.
Composite Object	An object that contains other objects. This is a result of the class for the object being an aggregate to the classes of the objects contained in the composite object.
Composite State	A state that contains other, less complex states.
Composition	An aggregation relationship in which the component is only part of one and only one aggregate at any given time. This implies that the component is not shared.
Composition measurement	An instance of measures for the amount of reuse based on composition reuse.
Composition reuse	Reuse of an artefact by using part or all of the artefact itself in another artefact. For example reuse of a customer class in a UML class diagram from a different class diagram developed in a previous project.
Compound Identifier	A unique identifier for a model element contained in a software model set.
Concept Absence	A literature source deemed relevant that does not include a conceptual term.

Conceptual Term	The number of concepts classified in a given vocabulary area.
Conceptual Vocabulary Component	The conceptual vocabulary component classifies concepts and their representation based on different vocabulary areas in a research surveying tool.
Construction Activity	An activity in a method process that is part of the construction process
Construction Process	Part of a software process that contributes to construction of the final product that can be used by users.
Core Concepts in Meta-Modelling	The core concepts in meta-modelling are Meta-Level Component (MLC), Meta-Level (MetLn), Interpretation or Meaning (IMea), Type and Instance, mn Model, Definition Method 9DefM), and Base Definition Method (Base DefM).
Coverage indicator	A hypothesis measure used to determine how successful the measurement framework is at measuring the amount of reuse with different software model types. These indicators are used primarily to determine the level of support for hypotheses H1 and H2.
Data collection instruments	Basic format of data collected in the experiment.
Data specifications	Data or requirements that are used prior to execution of the experiment.
Definition Activity	An activity in the method process where the method process itself is modified or defined. Definition activities are not construction activities or evaluation activities.
Definition Location (DefL)	A meta-level component that contains other meta-level components.
Definition Method (DefM)	A systematic specification of a meta-level component. Meta-level components can be used as definition methods for other meta-level components at lower meta-levels.
Degree of Cognition (Software Reuse)	The level of expertise used to support software reuse.
Dependency (Literature Classification)	A feature in a framework that relates different concepts to each other by means of some sematic term. For example, blood cells can be classified into white blood cells and red blood cells. Red blood cells are a kind of blood cell. Classification dependency exists between the concept of blood cells and red blood cells.
Dependent variables	Any data that is affected by independent variables. Dependent variables are usually related to data collection instruments.

Descriptive Model (Measurement)	A descriptive model measures attributes that currently exist. For example, the size of a class diagram.
Diagram	A notation or representation for a model. For example, a class diagram for a class model. This includes the icons and the rules for assembling icons to form a diagram.
Dimension (Literature Classification)	A feature in a framework to classify literature using mutually exclusive areas. For example, systems in the human organism include the nervous system and cardiovascular system.
Domain	The domain related to a particular software product. For example, finance or manufacturing.
Dynamic Part (Meta-Model Architecture)	The meta-level components in a meta-model architecture that can be modified, added, or deleted. In the measurement framework software model types classified using the software model type classification meta-level component contribute to the dynamic part of the meta-model architecture.
Encapsulation	Things are encapsulated when they are combined into a single larger unit. Some authors use this term as a synonym with information hiding. This thesis chooses to distinguish between encapsulation and information hiding.
Entity (Measurement)	Some thing that is measured via its attributes. For example, a class diagram (the entity) has the attribute of size.
Evaluation Activity	An activity in the method process that is part of an evaluation process.
Evaluation Process	Part of a software process that evaluates the use of a construction process and its effectiveness.
Event	A name given to a small period in time that makes a change of some kind. For example, the start of a race is an event. In state models. Events can cause a change in state and are linked to transitions.
Extensions to Meta-Modelling	The extensions to the core concepts in meta-modelling are Definition Location (DefL), Base Definition Location (Base DefL), Meta-Level Component Descriptor (MLCDesc), and Meta-Level Component Dependency (MLCDep).
External Attribute (Measurement)	Attributes that reflect some kind of outcome or end result related to an entity, for example, budgeted versus actual costs associated with a particular software process.
External reuse	Reuse of a software artefact that was *not created* as a result of the current project.
Formation	A phase in the measurement process where measures are defined and/or selected at the start of a software project.

Formulation	A synonym for formation.
Generated Model	The model that is generated from the source model in generation reuse. For example, C++ code is generated from a class diagram.
Generation measurement	An instance of measures for the amount of reuse based on generation reuse.
Generation reuse	Reuse of a software artefact by using it to generate or create part or all of another software artefact. This is done using some kind of transformation. For example, generation of C++ code from a UML class diagram.
Guard	A condition or requirement that must be true before a transition can take place. For example, a car engine can go from the off state to the on state if the car battery has a sufficient charge.
Homonym	A homonym exists when two conceptual terms share a common representation term.
Hypothesis analysis report	An analysis report for a single hypothesis (H0, H1, H2, H3, H4). Results for a single hypothesis are usually related to results from more than one methodology phase.
Hypotheses tested	The hypotheses under investigation in a methodology phase.
Hypothesis measure	A measure based on results of an experiment in a methodology phase (the collection process). Hypothesis measures *are not* measures of reuse. Hypotheses measures are base indicators, coverage indicators, or limitation indicators.
Identifier Part	A set of characters that help to identify an individual model element. An identifier part is an instance of an identifier part type.
Identifier Part Type	A part of a model element type that uniquely identify its instances. An identifier part type has a number of instances called identifier parts.
Implementation	A software artefact used to automate the use of a software model type in the software process.
Independent variables	Any data that affects outcomes of the experiment and dependent in a methodology phase. Independent variables are usually related to data specifications.
Indicator (Assessment Framework)	A synonym for hypothesis measure.
Information Hiding	Hiding part of the structure of some entity to support abstraction. For example, in objects the structure of attributes remains hidden.

Inheritance	A "kind of" relationship between classes that defines sub-groups of objects. For example, the class vehicle can be subdivided into the classes car and truck. The class truck is a kind of vehicle.
Instance	An object is an instance of some class. For example, a Toyota is car. The Toyota is the object, and Toyota is an instance of the class car.
Internal Attribute (Measurement)	Attributes that capture intrinsic qualities of an entity, for example, the number of stages in a software process.
Internal reuse	Reuse of a software artefact that *was created* as a result of the current project.
Interpretation (Measurement Process)	The interpretation of data that is analysed. Measures of the amount of reuse are analysed to determine reuse activity and reuse benefit.
Interpretation ort Meaning (IMea)	The description of a meta-level component that may include how it is used and its role in a meta-model architecture.
Iteration	A repetition of phases, subphases, or steps.
Kind of software model	A phrase with the same meaning as software model type. A software model type is a kind of software model.
Library Model	The product that is reused in composition reuse.
Limitation indicator	A hypotheses measure used to identify limitations of the measurement framework when measuring the amount of reuse for different software model types. These indicators are used primarily to determine the level of support for hypotheses H3 and H4.
Link	A relationship between objects based on the relationship between their respective classes.
Literature Source (Data Classification)	A document that describes a software model type.
Literature Source (Literature Classification)	A document that uses or defines a conceptual term.
Literature Source Reference	A literature source that uses or defines a conceptual term.
Maturity (Literature Classification)	Maturity or field maturity refers the progress made in some area of research.
Measurement (Method Area)	The aspect or component of a software methodology that describes the measurement process and measurement models.

Measurement Framework	A meta-model architecture designed to measure the amount of reuse for different software model types. The measurement framework is the subject of the experiments based on the hypotheses. The experiments are described in the assessment framework.
Measurement Process	The process used to define, collect, and use measures in the software process.
Measurement testing methodology phase	An experiment designed to see if it is possible to measure the amount of reuse for different software models using the measurement framework.
Message	The use of an operation in a supplier object by a client object. Sometimes referred to as a request.
Meta-Level (MetLn)	The level of a meta-level of a meta-level component.
Meta-Level Component	A component in a meta-model architecture that is part of a meta-model or mn model.
Meta-Level Component Dependency (MLCDep)	A dependency that a meta-level component has with other meta-level components.
Meta-Level Component Descriptor (MLCDesc)	A meta-level component that describes other meta-level components.
Meta-meta-model	An m2 model.
Meta-model	A model that defines other models.
Meta-Model Architecture	A framework based on meta-modelling. In software design, a meta-model architecture is an architectural style based on meta-modelling.
Method Area	An aspect or component that comprise a software methodology. There are three kinds in this thesis, namely, process, product, and measurement.
Method Levels	A series of layers used to classify and modify software methodologies. This is one approach to tailoring of software methodologies.
Method process	The structural elements of a process. This includes, steps, phases, activities, step and phase order, and products used on the process. Construction and evaluation processes are different kinds of processes, but both are structured along the lines of a method process.
Method Process Model Diagram	A diagram used to represent the method process.

Methodology Phase	Usually a phase in the assessment framework in this thesis methodology. Sometimes this term may need to be interpreted in context. The other interpretation would be a phase in a method process.
Methodology Phase (Assessment Framework)	An experiment in this thesis designed to find evidence to support or not support one or more hypotheses. Methodology phases must not be confused with software methodologies.
Methodology Phase Analysis report	An analysis report for a methodology phase.
Methodology Phase Description Template	A template used to systematically specify a methodology phase. The template has three parts, these are the collection process, variables, and the analysis process.
mn Model	A meta-model at level n. All of the meta-level components in this meta-model are at the same meta-level (n). For example, a meta-meta-model is another name for an m2 model. The meta-level of all meta-level components in the m2 model are equal to 2, that is, MetL2.
Model (Measurement)	A system used to measure an attribute of an entity. For example, the size of a class diagram can be measured using the system of by counting the number of classes.
Model Element	A part of a software model that models something in the real world. A model element is an instance of a model element type.
Model Element Type	A part of a software model type that denotes a kind of model element that instances of the software model type can contain. A model element type has a number of instances called model elements.
Model for Research Surveying	A meta-framework used to form research surveying tools for different research areas.
Modelling Concepts	Concepts used to define a software model. For example, classes and associations are modelling concepts in class models.
Multiple Inheritance	Multiple inheritance exists when a class is a subclass of two or more superclasses.
Object	A thing that has a name either abstract or real. For example, a Toyota is an object, so is a factorial function.
Object Diagram	The notation used to represent an object model.
Object Model	A software model based on the object-oriented paradigm that captures the state of a set of objects at a moment in time.

Object-Oriented Methodology	A software methodology that uses the object-oriented paradigm for modelling.
Object-Oriented Paradigm	A modelling paradigm that models the world in terms of objects, their relationships, and their behaviour.
Operation	The functions that an object can perform.
Organisation	An organisation that develops software using a software methodology.
Package	A notation icon that is used to manage complexity by representing a number of modelling concepts as one icon in a diagram. For example, in class diagrams packages can be used to represent a number of classes and relationships.
Part	Some thing or system that contains other parts. For example, a car (the whole) contains an engine and wheels (the part). Used as a synonym for component.
Percent Not Generated	The amount not generated expressed as a percentage of the complete model.
Percentage Added in Application Model	The amount added expressed as a percentage of the application model.
Percentage Contribution to Complete Model	The amount generated expressed as a percentage of the complete model.
Percentage of Library Model Not Reused	The amount not reused expressed as a percentage of the library model.
Percentage of Reuse in Application Model	The amount reused expressed as a percentage of the application model.
Percentage of Waste Generation	The waste generated expressed as a percentage of the generated model.
Phase	The name given to a procedure with some aim or purpose. The phase is usually represented as a task on a project plan.
Phase Diagram	A diagram used to represent the phases in a method process.
Phase Order	The order or sequence in which a number of phases can be performed.
Philosophical Principles	The foundation of a paradigm for a software methodology.
Polymorphism	The ability of an entity to change is form. In object-oriented methodologies, objects are polymorphic if they can change their class.

Predictive Model (Measurement)	A predictive model estimates attributes of an entity that do not currently exist. For example, the size of a C++ program based on a class diagram.
Principle of Reuse	The basic approach to reuse by either creating products composed from other products (composition reuse), or generating products that are reused in other products (generation reuse).
Process	A process represents the actions performed to produce products. Processes are entities that can be measured. Formally, a process of development based on the method process.
Product	A product is something produced and used in a process. Products are a kind of entity that can be measured.
Project	The development of a particular software product using a software methodology.
Relationship	A set of concepts that connect other concepts. The existence of a relationship requires the existence of other concepts that participate in the relationship. For example, aggregation is a relationship between two classes (the other concepts).
Representation Term	The number of terms used to represent a set of concepts.
Research Surveying Tool	A framework or model used to classify literature for a research area of interest to assess its maturity.
Resource	Resources are used to support the process. Resources are a kind of entity that can be measured.
Reuse	The practice of using the same entity in two different products. The products are usually related. For example, 60% of the parts in an Su-27 can also be used in the Mig-29. The design for these parts in the Su-27 are reused in the Mig-29.
Reuse activity	Steps and actions performed to reuse software.
Reuse Approach	An synonym for the principle of reuse.
Reuse benefit	Any benefits obtained as a result of reuse activity.
Role	The role given to class participating in an association relationship. For example, Jim manages jack, where Jim is a manager and Jack is an Employee. The association is manages. The class manager has the role of team leader.
Size	The size of a software model.
Size of Application Model	The size of the application model in composition reuse.

Size of Complete Model	The size of the complete model in generation reuse.
Size of Generated Model	The size of the generated model in generation reuse.
Size of Library Model	The size of the library model in composition reuse.
Size of Source Model	The size of the source model in generation reuse.
Software Asset	Any software artefact that was reused and provided some kind of reuse benefit.
Software Developer	A person who develops software using a software process.
Software Methodology	Anything that is related to a process to develop software.
Software Model (Informal)	A set of modelling concepts used together for a software model.
Software Model (Reuse)	An instance of a software model type.
Software Model Classification	A meta-level component in the measurement framework that classifies different software models using a composition hierarchy based on their software model type, model elements, and identifier parts.
Software model classification methodology phase	An experiment designed to see if it is possible to classify different software models and measure their size using the measurement framework.
Software Model Set	A mathematical set that defines the model elements contained in a software model An instance of the software models set.
Software Model Set Theory	A meta-level component in the measurement framework that classifies different software models using set theory.
Software Model Type	A formal classification of modelling concepts used to model software. For example, a class model can be classified as a software model type.
Software Model Type Classification	A meta-level component in the measurement framework that classifies different software model types using a composition hierarchy based on model element types and identifier part types.
Software model type classification methodology phase	An experiment designed to see if it is possible to classify different software model types using the measurement framework.
Software Model Type Set	A mathematical set that defines the model element types contained in a software model type. An instance of the software model types set.

Software Model Type Set Theory	A meta-level component in the measurement framework that classifies different software model types using set theory.
Software Model Types Set	An infinite mathematical set that defines the structure of all software model type sets.
Software Models Set	An infinite mathematical set that defines the structure of all software model sets.
Software Process	A process used to develop software. In this thesis "process" and "software process" are informally synonymous.
Software Project	A project to develop a specific software artefact using a software process.
Software reuse	The use of software artefacts in other software artefacts. For example reuse of a linked list in two program functions.
Source Model	The model that is used to generate another model in generation reuse.
Stage of Development	A major phase of development in a software process, usually part of the construction process. For example, analysis or design.
State	A value or set of values related to an object that changes over time. For example, a car engine can be on or off.
Statechart Diagram	The notation used to represent statechart models.
Statechart Model	A software model used in the object-oriented paradigm that captures the states and transitions for a class or system.
Static Part (Meta-Model Architecture)	The meta-level components in a meta-model architecture that cannot be changed. In the measurement framework the software model type classification and software model type set theory meta-level components are from the static part of the meta-model architecture.
Step	One or more activities that can be performed in parallel. Phases and subphases are ultimately broken down into steps.
Step Diagram	A diagram used to represent the steps and step order that constitute a phase or subphase in a method process.
Step order	The order or sequence in which a number of steps can be performed. Step order determines subphase and phase order.
Subclass	A class that participate in an inheritance relationship. For example, the class vehicle can be subdivided into the classes car and truck. The class truck is a subclass of the class vehicle.

Subdivision (Literature Classification)	A feature in a framework used to subdivide dimensions. For example, the cardiovascular system can be subdivided into veins and arteries. Subdivisions from different dimensions intersect to form vocabulary areas.
Subdivision Component	The subdivisions component classifies subdivisions in a dimension of a research surveying tool.
Sub-hypothesis	A hypotheses that is part of a research hypothesis (H1 or H2). Sub-Hypotheses are designed to break down the problem defined in the hypotheses into manageable sub-problems.
Subphase	A phase that is part of another phase in a method process.
Subphase Diagram	A diagram used to represent the subphases in a method process.
Subphase Order	The order or sequence in which a number of phases can be performed.
Subsystem	A package in a class diagram that can be contained by other packages and subsystems and may also contain subsystems.
Superclass	A class that participate in an inheritance relationship. For example, the class vehicle can be subdivided into the classes car and truck. The class vehicle is a superclass of the class truck.
Supplier	An Object that is a receiver of messages from a client. The Supplier performs one or more services requested by the client. Clients are connected to suppliers via a link.
Supplier Class	The class that supplies a service to another class called the client.
Synonym	A synonym exists for each representation term related to a conceptual term.
System Model	A complete model of a system. In this work a system model contains a number of models such as statecharts, class models, object models, and collaboration models.
System Model View	The representation of a software model that is part of a system model. For example, class models are represented using class diagrams.
Tailoring of Software Methodologies	The methods and techniques used to modify or adjust a software methodology.
Tailoring Phenomena	Tailoring of a software methodology to make it more effective in an organisation, a domain, or a project.
Traceability Reuse	Reuse of software artefacts based on their traceability.

Transition	A transition points to the new state that occurs as a result of an event. Transitions are associated with events. Transitions are part of statechart models and statechart diagrams.
Type and Instance (Meta-Modelling)	A meta-level component defines a type at its meta-level with instances of itself becoming types at the next lower meta-level.
Unified Modelling Language	A set of concepts with a notation for expressing models based on the object-oriented paradigm.
Variables	Any variables identified in the data from the collection process that are quantifiable. Variables are composed of independent and dependent variables.
Vocabulary (Literature Classification)	Concepts named in a framework that classifies literature. For example, literature on object-oriented methodologies includes the concept of a class. Concepts can be separated into their meaning and representation. For example, the concept of a class is represented as object class or object type and refers to a collection of objects with common attributes and operations.
Waste Generated	The part of a generated model that is not reused through generation reuse.
Whole	Some thing or system that contains other parts. For example, a car (the whole) contains an engine and wheels (the part). Used as a synonym for aggregate.