Using Model Driven Engineering to Reliably Automate the Measurement of Object-Oriented Software

Jacqueline A. McQuillan B.Sc. (Hons)

NUI MAYNOOTH

Ollscoil na hÉireann Má Nuad

Dissertation submitted in partial fulfillment of the requirements for candidate for the degree of

Doctor of Philosophy

Department of Computer Science,
National University of Ireland, Maynooth,
Co. Kildare, Ireland.

Supervisor: Dr. James F. Power

March, 2011

Contents

1 Introduction **1**
1.1 Background . . . 1
1.2 Motivation . . . 2
1.3 Goals and Approach . . . 4
1.4 Thesis Overview . . . 5

2 Background and Related Work **7**
2.1 Software Metrics . . . 7
2.1.1 Chidamber and Kemerer Metrics . . . 9
2.1.2 Coupling Metrics . . . 11
2.1.3 Cohesion Metrics . . . 13
2.1.4 The Elusive Goal of Precise Metric Definitions . . . 15
2.2 Model Driven Engineering . . . 16
2.2.1 Models and Metamodels . . . 17
2.2.2 Model Transformations . . . 18
2.2.3 Metamodelling Architectures . . . 19
2.2.4 Eclipse Modelling Framework . . . 23
2.3 Metamodelling and Software Metrics . . . 24
2.4 Software Testing in Model Driven Engineering . . . 28
2.4.1 Test Case Generation . . . 29
2.4.2 Test Adequacy Criteria . . . 30
2.4.3 Test Oracle Construction . . . 33
2.5 Summary . . . 33

3 **Towards an MDE Approach to Software Measurement** **35**
3.1 Introduction 35
3.2 Defining Metrics at the Meta-Level 37
3.3 dMML: Tool Support for Defining Metrics at the Meta-Level 40
3.3.1 Overview of dMML 43
3.4 Using the Approach to Define and Calculate Software Metrics . . . 45
3.4.1 An Illustration using the UML 46
3.4.2 An Illustration using Java 49
3.5 Discussion 53
3.6 Summary 54

4 **A Metamodel for the Measurement of Object-Oriented Software** **56**
4.1 Introduction 56
4.2 The Measurement Metamodel 57
4.2.1 Structure of the Metamodel 57
4.2.2 Defining Metrics using the Metamodel 61
4.3 A General Approach to Analysing MOF Metamodels 63
4.3.1 The Alloy Language and Analyser 63
4.3.2 Overview of the Analysis Approach 67
4.4 Development and Analysis of the Measurement Metamodel 69
4.4.1 Applying the Approach to the Metamodel 69
4.4.2 Analysing the Metamodel using Alloy 70
4.4.3 Discussion 72
4.5 Automatic Test Case Generation for MOF Metamodels 72
4.5.1 Test Case Generation Approach 73
4.5.2 Test Case Generation for the Measurement Metamodel . . . 75
4.5.3 Testing the Implementation of the Metrics 75
4.5.4 Discussion 80
4.6 Summary 81

5 **Testing Model Transformations for the Measurement Metamodel** **83**
5.1 Introduction 83
5.2 Model Transformation Languages 84
5.2.1 The Atlas Transformation Language 85
5.2.2 Integrating ATL with dMML 87

5.3 Measurement of UML Class Diagrams: The *UMLClassDiag2Measurement* Transformation 88
5.3.1 Source and Target Metamodels 88
5.3.2 The Transformation Rules 89
5.4 Measurement of Java programs: The *Java2Measurement* Transformation 92
5.4.1 Source and Target Metamodels 92
5.4.2 The Transformation Rules 92
5.5 Transformation Testing 95
5.5.1 ATL Coverage Criteria 95
5.5.2 Testing Strategy 97
5.6 Testing the Model Transformations 98
5.6.1 Testing the *UMLClassDiag2Measurement* Transformation . 99
5.6.2 Testing the *Java2Measurement* Transformation 105
5.7 Discussion 109
5.8 Summary 111

6 Concluding Remarks **113**
6.1 Contributions 113
6.2 Future Work 115
6.2.1 Improvements to the Measurement Approach 115
6.2.2 Further Applications of the Measurement Approach 116
6.3 Summary 119

Appendices **120**

A Measurement Metamodel Specification **120**
A.1 Attribute 121
A.2 BuiltIn 121
A.3 Class 121
A.4 FormalParameter 126
A.5 Invocation 127
A.6 InvocationType 127
A.7 Method 127
A.8 MethodType 131
A.9 NamedElement 132

A.10 Type 132
A.11 UserDefined 132

B Requirements for the Model Transformation Language 133

Bibliography 136

List of Figures

1.1 Overview of the software measurement approach. *The approach involves developing a measurement metamodel and a set of model transformations for converting software models to instances of the measurement metamodel. Metrics are then defined in OCL over the measurement metamodel and a measurement tool is automatically created that can be used to calculate the defined metrics for any software model.* 5

2.1 A sample UML class diagram and simplified extract from the UML metamodel. *This shows a class diagram representing a simplified University system along with a simplified extract of the UML metamodel [OMG07b] showing the parts relevant for modelling this class diagram.* 18

2.2 Overview of the model transformation process. *An overview of all the various different concepts involved in performing a model transformation.* 19

3.1 IAN Metric Definition. *This OCL code presents the* IAN *metric as defined by Baroni [Bar02]. It is part of a larger set of measures called FLAME and specifies that the metric computes the number of inherited attributes for a* `Classifier` *element of the UML 1.3 metamodel. Any auxiliary operations used in this definition can be found in [Bar02].* 37

3.2 Extension to the language metamodel. *This package diagram shows the definition of a set of metrics as a separate package, with a dependency on (meta)classes from the language metamodel.* 39

3.3 Representing metamodels as models. *This diagram shows how the UML metamodel and UML class diagram at the M2 and M1 layers of the OMG metamodelling hierarchy can be represented at the M1 and M0 layers. Although diagrams (a) and (c) on inspection look identical, please note that the elements in (a) are elements from the UML metamodel and reside at the M2 level and the elements in (c) are elements from a class diagram and reside at the M1 level.* 41

3.4 dMML - An environment for the definition of software metrics. *This system overview diagram shows the main inputs to and outputs from the dMML tool, which is implemented as an Eclipse plug-in.* . 43

3.5 The use of dMML to define and calculate metrics for UML class diagrams. *This figure shows the two phases of our system: metric definition, centered on the dMML tool, and metric calculation, achieved by our UML metamodel instantiator program and an automatically generated measurement tool.* 47

3.6 Excerpt from the UML2.0 metamodel. *This figure shows some of the main classes and relationships from the UML2.0 metamodel that are used in the OCL definitions [OMG05b].* 47

3.7 NOC Metric Definition using the UML metamodel. *This OCL code defines the* NOC *metric from the CK metrics set, and is part of a larger definition of the whole CK metric set which we have implemented using dMML.* 48

3.8 The use of dMML to define and calculate metrics for Java programs. *This figure shows the two phases of our system: metric definition, centered on the dMML tool, and metric calculation, achieved by our Java-to-DMM program and an automatically generated measurement tool.* 50

3.9 Excerpt from the Dagstuhl Middle Metamodel. *This figure shows some of the main classes and relationships from the DMM that are used in our OCL definitions [LTP04].* 51

3.10 RFC Metric defined using the DMM. *This OCL specification defines an operation to calculate the RFC metric for a class, as well some auxiliary operations. The entities used in the definition are from the DMM [LTP04].* 52

3.11 NOC Metric Definition using the DMM. *This OCL specification defines an operation to calculate the NOC metric for a class. This definition is provided here for comparison with the NOC metric for UML class diagrams shown in Figure 3.7.* 53

4.1 The measurement metamodel. *This figure shows the main classes and relationships in the measurement metamodel depicted as a UML class diagram. For clarity the NamedElement class has been omitted from this diagram.* 60

4.2 Definition of the *CBO* metric in OCL using the measurement metamodel. *This OCL code defines the CBO metrics from the Coupling metric set, and is part of a larger definition of a set of Coupling metrics which has been defined and implemented using the measurement metamodel.* 62

4.3 Alloy specification for the Queue model. *This is an example of an Alloy specification and defines the Queue data structure.* 65

4.4 Output from the Alloy Analyser produced during the analysis of the Queue specification *On the left is a model instance that satisfies the predicate `show` and an the right is a counterexample to the assertion `allNodesBelongToSomeQueue`.* 66

4.5 Overview of the approach to analysing the MOF-compliant metamodel. *The elements in the system are enclosed by a dashed red line. The input, shown on the left, is the metamodel and its constraints expressed using UML and OCL.* 67

4.6 Alloy signature for the element `Class` of the measurement metamodel. *This is a representation of the element `Class` in the Alloy specification language.* 70

4.7 An example of a constraint on the measurement metamodel written in both OCL and Alloy. *This constraint states that if a class does not have any parents then it cannot have any overridden methods.* . 71

4.8 Overview of the approach to test case generation for a MOF-compliant metamodel. *The elements in the system are enclosed by a dashed red line. The input, shown on the left, is the metamodel and its constraints expressed using UML and OCL. The outputs, shown at the bottom, are the metamodel implementation and test instances in Java.* 73

5.1 The sample *FamiliesToPersons* ATL transformation. *This piece of ATL code shows a sample transformation that convert a list of families into a list of people and consists of a transformation rule and a helper operation.* 86
5.2 The source and target metamodels of the *FamiliesToPersons* Transformation. *This figure shows the metamodel Families used as the source metamodel in the transformation and the metamodel Persons used as the target metamodel of the transformation.* 86
5.3 Overview of the aproach used to calculate metrics for class diagrams. *The input to the system is a UML class diagram, and the outputs are the metrics for the UML model. The system itself is delineated by a dashed box.* 87
5.4 Example of a transformation rule in ATL. *This piece of ATL code shows a transformation rule,* `FieldProperty2Attribute`*, and an associated helper operation called* `isField`*.* 91
5.5 Example of a transformation rule in ATL. *This piece of ATL code shows a transformation rule,* `Constructor2Method`*, and two associated helper operations called* `isAbstract` *and* `isPublic`*.* 94
5.6 Overview of the approach used to test our model transformation. *The input to the approach is the set of source models and the model transformation and the output is a tested model transformation.* . . . 97

6.1 An overview of applying the CK metrics to UML models. *This figure reviews the diagrams in a UML model that can contribute to calculating the CK metrics [MP07].* 118

List of Tables

2.1 The Four Layer Metamodel Architecture. *This table shows the standard four-layer hierarchy, using the UML modelling as an example (p19 of [OMG07a]).* 20

4.1 Summary of the size of the measurement metamodel. *This table gives a summary of the size of the metamodel in terms of the different types of MOF element that constitute the metamodel.* 58

4.2 Summary of implemented metrics. *This table lists all 44 metrics that were defined and implemented using the measurement metamodel.* 61

4.3 Groups of generated test cases. *There were four main groups of test cases, generated by varying the settings for Alloys model generator. The number of test cases in each group is shown in the final column.* 75

4.4 Line/Branch coverage excluded from the coverage targets. *This table lists three kinds of code excluded from the coverage targets, along with the percentage of lines/branches for each kind.* 77

4.5 A breakdown of the metamodel coverage for each of the test groups in Table 4.3. *The numbers presented for each group represent the cumulative coverage achieved, including the previous test groups.* . 78

4.6 Test cases in the reduced test suite. *This table lists the 14 test cases in the reduced suite, along with the cumulative coverage figures for the two coverage criteria.* 79

5.1 A summary of the test cases. *This table lists the class diagrams from the UML2 Tools project [Eclf] (release 0.8.0 of 11 June 2008). In future tables we refer to these models by number only; this table can be used to refer back to models in the UML2 Tools distribution and UML 2 superstructure specification [OMG07b].* 100

5.2 A summary of the percentage coverage for each of the test cases in Table 5.1. *This table summarises the coverage for each of the 20 UML models in the* UMLClassDiag2Measurement *transformation test suite.* 102

5.3 A breakdown of the overall cumulative branch coverage data for all 20 UML models. *This table splits the branch instructions into four categories based on the degree to which they were covered during the transformations.* 103

5.4 Metric results for the test suite. *This table lists some basic (and easily verifiable) object-oriented metrics for each of the 20 UML models from Table 5.1 and test cases 21 and 22, the additional models created to increase the coverage to the maximum possible.* 104

5.5 A summary of the percentage coverage for each of the Java packages generated from the UML models in Table 5.4 *This table summarises the coverage for each of the 22 Java packages in the Java2Measurement transformation test suite.* 107

5.6 A breakdown of the overall cumulative branch coverage data for all 22 Java packages. *This table splits the branch instructions into four categories based on the degree to which they were covered during the transformations.* 108

B.1 List of model transformation languages. *This table summarises all of the model transformation languages and tools that were considered for inclusion in the measurement approach and how they evaluate according to our requirements.* 135

List of Acronyms

ATL	Atlas Transformation Language
CK	Chidamber & Kemerer
COTS	Commercial Off-The-Shelf
CWM	Common Warehouse Metamodel
DIT	Depth of Inheritance Tree
dMML	Defining Metrics at the Meta Level
DMM	Dagstuhl Middle Metamodel
DSL	Domain Specific Language
MDA	Model Driven Architecture
MDD	Model Driven Development
MDE	Model Driven Engineering
MM	Measurement Metamodel
MOF	Meta Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
OO	Object Oriented
PIM	Platform Independent Model
PSM	Platform Specific Model
RFP	Request For Proposal
UML	Unified Modelling Language
WFR	Well-Formedness Rule
XMI	XML Metadata Interchange
XML	eXtensible Markup Language
XSD	XML Schema Definition
XSLT	eXtensible Stylesheet Language Transformation

Declaration

I confirm this is my own work and the use of all material from other sources has been properly cited and fully acknowledged. Part of the work in this thesis has been presented in the publications listed in the preface of this thesis.

Signed:_________________________________

Jacqueline A. McQuillan
National University of Ireland, Maynooth
March 7, 2011

Acknowledgements

This PhD would not have been possible without the help and support of three people in particular and I would like to begin by extending my gratitude to them - to my supervisor Dr. James Power, Tom and Sue, a very big thank you.

I would like to thank my PhD supervisor, Dr. James Power, first for accepting me as a PhD student and giving me the opportunity to complete this PhD and second for all his time, effort, patience and guidance that he has invested and provided throughout the course of this PhD. I am extremely grateful.

A very special thanks to Tom, for all his support, encouragement and advice. Thanks for being there for me through all the hard times, for putting up with me during my times of self doubt and crankiness while writing this thesis and still being there for me at the end of it all. I am very very lucky.

Special thanks are also due to Sue for being a true friend and always being there when I needed some good advice. I have no doubt that I would not have reached this point without it. Thanks are also due for taking the time to proof-read this thesis.

I would also like to thank Aidan for proof reading this thesis but more importantly for being a great friend over the years and keeping me entertained with all those hilarious jokes. I look forward to hearing many many more.

I would also like to extend my thanks to Professor Alan Mycroft for providing me with a place in his lab at the Computing Lab, Cambridge University in 2006/2007. I really appreciate the generosity and kindness that he showed during my time there, in particular the many times that he took to discuss my research.

Many thanks to all my friends who have supported me during this time and had faith in my abilities to finish this PhD, in particular I'd like to thank Fiona, Marian and Sandie. I promise to do a better job at keeping in touch now that this is done.

Finally, I'd like to thank my family, Mam, James and Gerard for supporting me in this endeaour over the last couple of years. For all the times they have asked "Are you finished yet?", I'm happy to finally be able to say "Yes. I'm finished!"

For my Mam.

Preface

Parts of this thesis have been presented in the publications listed below.

Jacqueline A. McQuillan and James F. Power. Test-driven development of a metamodel for the measurement of object-oriented systems. Invited for publication in *Software Testing, Verification and Reliability (STVR)*. Under review. 2008.

Jacqueline A. McQuillan and James F. Power. A metamodel for the measurement of object-oriented systems: An analysis using Alloy. In *Proceedings of the 1st IEEE International Conference on Software Testing, Verification and Validation (ICST)*. pp. 288-297. Lillehammer, Norway. 9-11 Apr 2008.

Jacqueline A. McQuillan and James F. Power. On the application of software metrics to UML models. In *Models in Software Engineering - Workshops and Symposia at MoDELS 2006, Reports and Revised Selected Papers*. Springer Lecture Notes in Computer Science. Vol. 4364. pp. 217-226. 2007.

Jacqueline A. McQuillan and James F. Power. Some observations on the application of software metrics to UML models. In *Model Size Metrics Workshop of the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS)*. Genoa, Italy, 1-6 Oct 2006.

Jacqueline A. McQuillan and James F. Power. Experiences of using the Dagstuhl Middle Metamodel for defining software metrics. In *Proceedings of the 4th International Conference on Principles and Practices of Programming in Java (PPPJ)*. pp. 194-198. Mannheim, Germany, 30 Aug-1 Sept 2006.

Jacqueline A. McQuillan and James F. Power. Towards re-usable metric definitions at the meta-level. In *PhD Workshop of the 20th European Conference on Object-Oriented Programming (ECOOP)*. Nantes, France, 3-7 July 2006.

Abstract

Software metrics have many important uses in software engineering, for example, assessing software quality or estimating the cost and effort of developing software. Many metrics have been proposed and new metrics continue to appear in the literature. Many of these metrics are incomplete, ambiguous and open to a variety of different interpretations making it difficult to create general metric tools. Furthermore, everytime a new software metric is defined the tools need to be updated to include the new metric. This makes it difficult to perform independent validation of empirical results and to investigate how different metrics relate to each other.

Model Driven Engineering (MDE) is an emerging approach to software development in which models are the primary focus. In this model-oriented approach, design artifacts such as Unified Modelling Language (UML) diagrams as well as implementation artifacts such as program code are considered as different models of a software system. Working at the model level provides a whole new set of constructs to be measured and recent research has moved towards new model-based metrics. Some of these new metrics and many existing metrics are applicable to a number of different models of a software system. In order to provide assurance that the same concepts are being measured from different software models, metrics need to be specified in a generic way that is not dependent on the particular model.

This thesis describes the development, implementation and testing of an MDE-based approach to the measurement of software models. This approach involves specifying software metrics using the Object Constraint Language (OCL) and a measurement metamodel and transforming all other models to this canonical metamodel. Using this approach only a single definition of a software metric is required and can be applied to different models of a software system thus helping to provide assurance that the same concepts are being measured from the different models. Furthermore, this approach eliminates the need for manual implementation of metrics tools as it supports the automatic generation of a measurement tool from the

metric definition. Finally, to ensure that this approach is reliable this thesis develops testing techniques for the domain of MDE and applies them to the measurement approach. These techniques are fundamental to the approach, including validation of the underlying measurement metamodel, model transformations and automatically generated measurement tool.

The main contributions of the work presented in this thesis are: a Meta Object Facility (MOF)-compliant measurement metamodel for coupling and cohesion metrics; the definition of standardised transformations from the UML and Java to this metamodel; testing techniques for use within the MDE, specifically approaches for analysing and testing metamodels, metamodel-based software and model transformations.

Chapter 1

Introduction

This research is primarily concerned with the development of a Model Driven Engineering (MDE)-based approach to software measurement. This chapter briefly introduces the areas of software metrics and MDE and provides the motivations for developing the measurement approach. The goals of this thesis are also presented along with an overview of the thesis structure.

1.1 Background

Software metrics provide a quantitative way to capture certain attributes or characteristics of a software system. They have many important uses in several areas of software engineering. One such area is determining software quality where several studies have demonstrated a correlation between software metrics and quality attributes such as fault-proneness and maintenance effort [BBM96, LH93]. They have also been proposed as a way to estimate the cost and effort of developing software [FP96]. Metrics are not limited to quality assessment but have also been used in other domains such as software evolution and software re-engineering.

In the domain of software evolution, metrics have been used to identify parts of a software system that are in need of refactoring as well as to determine if parts of a software system have already been re-factored and to assess the improvements in the quality of the software as it is re-factored and evolves [DDN00, LD02]. In the domain of software re-engineering and reverse engineering, metrics have also been used to determine software complexity and quality and also to identify parts of the

system that require re-engineering [LD02, Kol02].

MDE is an emerging approach to software development in which models are the primary focus. There are many aspects to software development that make it a difficult and labour intensive task. For example, software systems often need to communicate with other systems, when new technologies are developed systems need to be updated and there is also the problem of constantly changing requirements [WKB03]. MDE attempts to tackle these problems by providing an approach to software development in which models are used as the primary artifacts when developing software, often with the code generated automatically from these models. It aims to provide ready-made models and transformations that convert one model to another. Instead of developing and re-developing a software system as the requirements or technologies change, models are chosen, modified or extended and combined together with other models to create the system [Ken02, WKB03, Sch06]. As the software engineering community begins to adopt MDE as an approach to developing software, it follows that the development of software measurement tools will also adopt this approach. Therefore, it is logical to investigate if MDE principles can be successfully applied to automate the measurement of software.

One important and sometimes overlooked aspect of software measurement is ensuring that the process of measuring software is correct and reliable. For MDE to provide a suitable and adequate approach to software measurement, the resulting measurement approach must meet these criteria. One way to assess the correctness of software is by applying software testing techniques [Bei90]. However, due to the novelty of MDE there is a need for the development of testing techniques for this new domain [KAER06, BDTM+06]. In this thesis, we propose methods for carrying out software measurement using MDE techniques and also testing techniques to ensure that the measurement process is correct and reliable.

1.2 Motivation

Many metrics have been proposed and new metrics continue to appear in the literature regularly [FP96, CK94]. A fundamental problem however with these metrics is that many of them are imprecisely and ambiguously defined. This is as a result of there being no standard formalism or terminology for defining or expressing software metrics [BDW99, BBA02]. This allows for different interpretations of metric definitions by different researchers [BDW98, KHL01]. This makes it difficult to

create general metric tools and every time a new metric is defined the tools need to be updated with the new metric [ML02]. In a recent study by Lincke *et al.* several software metric tools were evaluated and compared. It was shown that all of the tools considered interpreted and implemented the object-oriented metrics differently which resulted in tool-dependent metric results [LLL08]. All these issues make it difficult to replicate experiments and perform independent validation of empirical results and also to investigate how different metrics relate to each other [BDW99, KHL01]. This hinders empirical validation of the metrics and ultimately their acceptance and adoption by both the academic and industrial software metrics community.

In an MDE approach to software development, a software system is created as a series of models at a number of different levels of abstraction and eventually an implementation is created. Many software metrics can be applied to a number of different models of a software system. Measurement inconsistencies can arise where a software metric is applied differently to different models of the same software system. Therefore, in order to provide assurance that the same concepts are being measured from these different models we need a way to specify the metrics in a generic way, independent of the particular model [ML02].

Many of the existing approaches to measuring software metrics involve the analysis of source code. As a result, it is not always clear how to apply existing metrics at the early stages of the software development process. With the development of software shifting to models rather than source code, particularly within the domain of MDE, research is required to investigate how software metrics can be measured from software models and prior to the implementation of the system. Being able to measure the metrics accurately from both models and source code is important for several reasons [MP07], including

- The quality of the system can be assessed in the early stages of the software life-cycle when it is still cost effective to make changes to the system.
- The implementation can be assessed to determine where it deviates from its design. This can be achieved by applying metrics to both the model and source code and comparing the results. Variations in the metric values may help to identify parts of the implementation that do not conform to its design.
- Evaluation of the correctness of round trip engineering tools can be performed. Again, applying the same metrics to both the models and source

code may help in identifying parts of the system that have been incorrectly forward or reverse engineered.

1.3 Goals and Approach

The main goal of this thesis is to apply the principles of MDE to develop an approach to software measurement that

- provides a standard terminology and formalism for specifying software metrics.
- supports the automation of the generation of measurement software directly from the metric specifications.
- is highly extensible and can easily incorporate new metrics as they appear as well as multiple definitions of the same metric.
- is language independent and provides assurance that the same concepts are being measured from the different models of the same software system.

A complementary goal of this work is to ensure that the approach is correct and reliable by developing methods and techniques for testing in the domain of MDE and using them to test the measurement approach.

In order to achieve the main goal we first develop an approach for specifying metrics that is based on language metamodels and the Object Constraint Language (OCL) [WK03] and develop a flexible and re-usable environment that automatically generates a measurement tool from the metric specifications. We then expand the approach in order to make it language independent by proposing the use of a language independent measurement metamodel for the definition of software metrics and that to calculate metrics for any model will involve transforming the model to this canonical measurement metamodel. With this approach only a single definition of a software metric is required and can be applied to different models of a software system, thus providing assurance that the same concepts are being measured from the different models.

Figure 1.1 presents an overview of this measurement approach which takes three inputs: a set of source models that conform to a source metamodel, a measurement

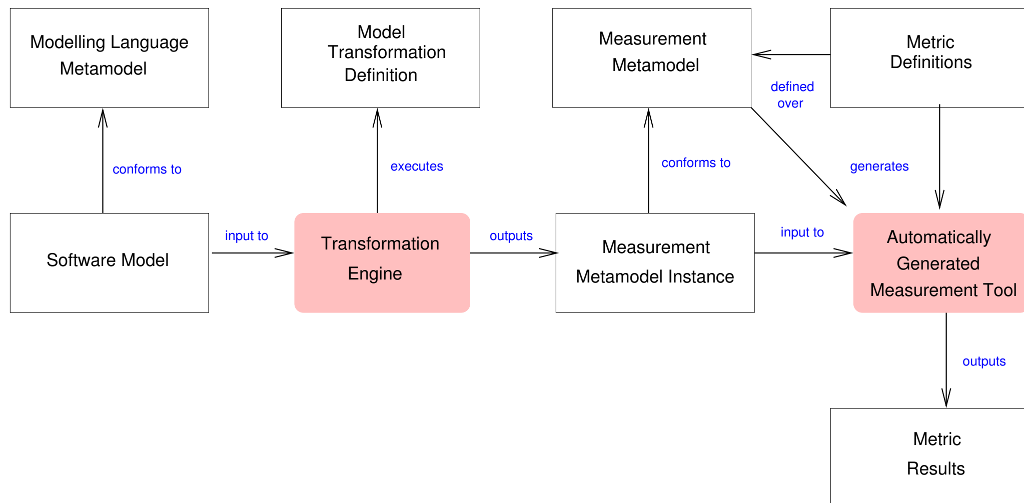

Figure 1.1: Overview of the software measurement approach. *The approach involves developing a measurement metamodel and a set of model transformations for converting software models to instances of the measurement metamodel. Metrics are then defined in OCL over the measurement metamodel and a measurement tool is automatically created that can be used to calculate the defined metrics for any software model.*

metamodel and a model transformation for transforming source models to measurement metamodel instances. Metrics are then defined as OCL queries over the measurement metamodel and a measurement tool is automatically created for evaluating the defined metrics over instances of the measurement metamodel. We limit the scope of our work to the domain of object-oriented coupling and cohesion measurement and develop a metamodel specifically for coupling and cohesion metrics.

Finally, to ensure that this approach is reliable we develop testing techniques for the domain of MDE and apply them to the measurement approach. These techniques are fundamental to the approach, including validation of the underlying measurement metamodel, model transformations and automatically generated measurement tool.

1.4 Thesis Overview

The remainder of this thesis is organised as follows. Chapter 2 provides an introduction to the areas of software metrics and MDE while also reviewing the relevant research in these fields and comparing it with the work presented in the thesis.

In Chapter 3 an approach to the specification of metrics that is based on language metamodels and the OCL is presented. The approach is illustrated using the

Chidamber and Kemerer (CK) metrics suite [CK91, CK94] and the Unified Modelling Language (UML) 2.0 and Java metamodels. Details of the dMML (Defining Metrics at the Meta Level) tool that supports this approach is also presented.

Chapter 4 is concerned with the development and testing of a measurement metamodel over which coupling and cohesion metrics can be expressed. This chapter presents a Meta Object Facility (MOF)-compliant metamodel for coupling and cohesion measurement and describes a systematic approach that can be used to both analyse MOF-compliant metamodels and to generate test data for any software based on a MOF-compliant metamodel.

The measurement approach is completed in Chapter 5 by introducing support for model transformations and transformation testing. It outlines how an existing transformation language, Atlas Transformation Language (ATL) [ATL] has been incorporated into the dMML tool to provide complete tool support for the measurement approach and describes an approach to testing model transformations. Two applications of the measurement approach to both UML class diagrams and Java programs are also described.

Chapter 6 summarises the work presented in this thesis, identifies its main contributions and presents a discussion of future work. This thesis also contains several appendices containing a full specification of the measurement metamodel and an outline of the requirements for choosing a model transformation language for the measurement approach. All further supplementary information including the metric specifications, metric results and software developed to support this work can be found on the disk accompanying this thesis.

Chapter 2

Background and Related Work

The overall goal of this thesis is to develop a reliable MDE-based approach to software measurement. Therefore, this thesis encompasses two main research areas: MDE and software metrics. The purpose of this chapter is to present the main concepts of these two areas and to review the work relevant to the research presented in this thesis.

This chapter is structured in three parts; the first part introduces the area of software metrics, presents the three sets of object-oriented software metrics that are used in this thesis and reviews the state of the art of existing approaches to the specification and definition of software metrics. In the second part of the chapter we introduce the area of MDE. As the thesis is concerned with developing a reliable approach to software measurement using MDE principles, the final part of the chapter reviews software testing research in the context of MDE. Parts of this chapter have been published in McQuillan and Power [MP06c, MP07].

2.1 Software Metrics

Software metrics provide a quantitative way to capture certain attributes or characteristics of a software system or product. They have many important uses in software engineering including determining software quality and estimating the cost and effort of developing software [FP96, BBM96, LH93]. Metrics are not limited to quality assessment but have also been used in other domains such as software evolution and software re-engineering [LD02].

Software metrics may be classified according to the entities that they measure, as either product, process or resource metrics [FP96]. Product metrics are measures of the artifacts or deliverables that are produced while developing the software. Examples of product metrics include those which measure attributes of complexity, size or maintainability of the software design or piece of source code.

Over the years, many software product metrics have been presented in the literature. Some of the earliest measures are those proposed for estimating software size and complexity, such the Source Lines of Code (SLOC) metric, Halstead's measures and McCabe's Cyclomatic Complexity [FP96, Hal77, McC76]. With the introduction of the object-oriented paradigm for developing software, the structure and organisation of the software shifted from procedures to classes and functions. Several of the existing measures were adapted to this new paradigm and many researchers proposed new metrics specific to the object-oriented paradigm. Such metrics include the suite of metrics proposed by Chidamber and Kemerer and Lorenz and Kidd [CK91, LK94]. As well as metrics being proposed for specific paradigms, researchers have proposed metrics specific to particular programming languages such as C++ [BDM97, EBC05].

More recently, research has been conducted to develop software metrics for pre-code artifacts such as the UML. One of the earliest sets of metrics proposed for UML models are those described by Marchesi who propose a set metrics that can be applied to class and use case diagrams [Mar98]. This metric set is composed of 19 metrics which count the various elements that can be found in a UML class and use case diagram. The motivation for developing these metrics is so that they can be used for estimating the effort, time and cost involved in developing a software system at the early stages of the development process. However, very little information is given on how effective these metrics are at fulfilling this purpose.

Genero *et al.* have proposed a set of metrics for assessing the structural complexity of class diagrams and have performed several experiments to empirically validate these metrics [GPC00, GJP02]. Again these metrics are primarily concerned with counting the elements in a class diagram and they show that the metrics have a positive correlation with the three sub-characteristics of maintainabilty: analysability, understandability and modifiability. Various other metrics have been proposed for class diagrams and a comparison of these metrics are presented by Yi *et al.* [YWG05], including those proposed by Marchesi and Genero *et al.*. The metrics are evaluated and discussed according to several criteria including the types

of relationships they consider, the complexity of the metrics themselves and the amount of empirical validation that has been performed to demonstrate the usefulness of the metrics.

Genero *et al.* have also developed a set of metrics for measuring the size and structural complexity of state-chart diagrams [GMP02]. A total of 5 metrics are proposed and are based on counting the number of elements in a statechart diagram (e.g. states, transitions etc.). The authors theoretically validate these metrics and also conduct a study to empirically validate them as early maintainablity indicators.

Kim and Boldyreff have defined a set of 27 metrics to measure various characteristics of a UML model [KB02]. However, the metrics are described informally and for some of these measures it is unclear which UML diagrams should be used to calculate the measures. Tang and Chen have also attempted to measure UML models by specifying how the CK metrics can be measured from UML diagrams [TC02]. They have developed an algorithm for computing the metrics from UML class, activity and communication diagrams.

As we have outlined, the evolution of software metrics has progressed from basic code metrics for procedural languages to object-oriented languages right up to design artifacts such as the UML. A number of these however are applicable to a number of different languages and levels of abstraction. We have selected a set of metrics taken from [CK94, BDW98, BDW99] which we believe can be usefully applied at different levels of abstraction and have specified and implemented them using our measurement approach. For completeness we describe these metrics in detail in the remainder of this section.

2.1.1 Chidamber and Kemerer Metrics

One of the most well known suite of object-oriented metrics is the one proposed by Chidamber and Kemerer [CK91, CK94]. The metrics were proposed to assess different aspects of an object-oriented design and consist of the following six metrics:

Weighted methods per class (WMC)

The weighted methods per class (WMC) metric is concerned with the structural complexity of a class in an object-oriented system. The metric is computed by summing the complexities of all the methods in a class. The authors of the metric have deliberately left the definition of a method's complexity open "in order to allow

for the most general application of this metric" [CK94]. For our purposes we have taken the complexity to be unity, thus reducing the WMC metric to a count of the methods in a class.

Depth of inheritance tree (DIT)

The depth of inheritance tree (DIT) for a class is the maximum length from the root of the inheritance tree to that class. Essentially it is a count of the number of ancestors of the class and is based on the principle that the greater the number of ancestors of a class then the greater the number of attributes and methods it is likely to inherit. This can increase the overall complexity of the class and make it more difficult to determine its structure and behaviour [CK91, CK94].

Number of children (NOC)

Number of children (NOC) is defined as the number of immediate subclasses subordinate to a class in the class hierarchy. This metric assesses the number of classes that are likely to inherit the attributes and methods of a class. As the number of immediate descendents of a class increases then the potential influence of that class also increases [CK91, CK94].

Coupling between object classes (CBO)

This is first defined as a count of the number of non-inheritance related couples with other classes [CK91] and is referred to as CBO' by Briand *et al.* [BDW99]. Two classes are coupled if the methods of one class reference attributes or invoke methods defined in the other class. This definition was later revised to include coupling due to inheritance [CK94] and is referred to as CBO by Briand *et al.* [BDW99]. High CBO values are more likely to have an adverse effect on the maintenance, reuse and testing of the overall system as changes or faults in one class have a greater probability to propagate to other parts of the system.

Response for a class (RFC)

Response for a class (RFC) is the size of the response set for a class, where the response set is defined as the set of distinct methods that can be invoked from that class. The RFC is calculated for a class `c` by adding the number of methods of `c`

to the number of external methods called by the methods of `c`. RFC refers to only those methods that are directly invoked by `c` where RFC' considers methods that are directly and indirectly invoked by `c`. The larger the RFC value the more complex the class is. This is because the more methods that are called in a class implies that more code must be examined in order to understand the class [CK91, CK94].

Lack of cohesion in methods (LCOM)

Lack of Cohesion in Methods (LCOM) metric for a class is summarised as the "degree of similarity of methods" and is adapted from Bunges notion of similarity which defines the similarity of things to be the set of properties that the things have in common [CK91, Bun77, BDW98]. It is first defined as the number of non-intersecting sets of methods based on the common usage of instance variables [CK91]. It is later defined as the number of method pairs in a class that use common instance variables minus the number of pairs of method that do not use any common variables [CK94].

2.1.2 Coupling Metrics

The notions of coupling and cohesion were first proposed by Stevens *et al.* when addressing the question of what makes a "good design". They define coupling as "the measure of the strength of association established by a connection from one module to another" [SMC74]. Yourdon and Constantine also describe the concepts of coupling and cohesion, describing coupling as the degree of interdependence between modules [YC79, FP96]. The notion of coupling has also been adapted to object-oriented systems [CY91, CK91]. In all cases, the general consensus is that low coupling is desirable as the more independent a module or class is, the easier it is to understand, maintain and re-use.

A considerable number of coupling metrics have been proposed in the literature. As part of their framework for coupling measurement, Briand *et al.* identify and describe a number of different object-oriented coupling measures [BDW99]. These measures include the CBO and RFC metrics proposed by Chidamber and Kemerer along with several other measures which we briefly explain in the remainder of this subsection.

Li and Henry propose two metrics, message passing coupling (MPC) and data abstraction coupling (DAC) for predicting the maintainability of an object-oriented

design. MPC is defined as "the number of send statements defined in a class" where send statements are considered to be the static invocations of a method. MPC only counts invocations of methods of other classes, not invocations of its own methods. DAC is defined as "the number of abstract data types (ADT) defined in a class", where an ADT is any class in the system. An ADT is defined in a class `c` if an attribute of `c` is of that type. Two interpretations of this metric are possible, one which counts the number of attributes in a class having an ADT and is referred to as DAC. The other interpretation, referred to as DAC' counts the number of distinct ADT's defined in a class [LH93, BDW99].

The Coupling Factor (COF) metric proposed by Abreu *et al.* is a count of the total number of client-server relationships between non-inheritance related classes in a system divided by the maximum number of such client-server relationships that can occur in the system [AGaE95]. This metric was proposed as part of a larger set of metrics for evaluating the quality of an object-oriented system. To facilitate the comparison of systems of varying sizes this metric is normalised to range between 0 and 1 [AGaE95, BDW99].

Lee *et al.* propose a number of metrics that are based on the flow of information through a program [LLWW95]. One of these metrics, Information-flow-based coupling (ICP), is a metric defined at the method level that counts the number of methods belonging to all other classes that are polymorphically invoked by the method, weighted by the number of parameters of the invoked method. It is possible to scale this metric to the class level by summing the ICP values of each of the methods in a class and similarly to the system level by summing the ICP values of all the classes in the system. Two further metrics are also defined, inheritance-based coupling (IH-ICP) and non-inheritance-based coupling (NIH-ICP). The IH-ICP metric considers couplings due to ancestor classes whereas the NIH-ICP metric only considers couplings due to unrelated classes. The ICP metric is simply the sum of IH-ICP and NIH-ICP [LLWW95, BDW99].

Briand *et al.* define three measures which capture the different types of interactions that can occur between two classes, class-attribute (CA), class-method (CM) and method-method (MM) interactions. CA-interactions occur from class `c` to class `d` when an attribute of class `c` is of type class `d`. CM-interactions occur from class `c` to class `d` when a newly defined method of class `c` has a parameter of type class `d`. MM-interactions occur from class `c` to class `d` when a method implemented in class `c` statically invokes a newly defined or overridden method of class `d`, or when

it receives a pointer to such a method. Using these three types of interactions they define a set of 18 coupling metrics [BDM97, BDW99].

2.1.3 Cohesion Metrics

Stevens *et al.* define cohesion as a measure of the degree to which the elements of a module belong together [SMC74, BDW98]. In parallel with coupling, the principle of cohesion has also been adapted to object-oriented systems [CY91, CK91]. High cohesion is desirable as this suggests that all the components are strongly related and are working together to perform a single function which it is proposed makes it easier to develop, maintain, and reuse a module or class and makes it less fault-prone [BDW98].

The definition of cohesion for object-oriented systems began with the initial proposal of the LCOM metric, followed by several proposed improvements of this metric by different authors [CK91]. Henderson-Sellers interpret the LCOM metric as the number of pairs of methods in a class `c` that have no common attribute references and this is referred to as LCOM1 by Briand *et al.* [HS96, BDW98]. Chidamber and Kemerer revise their LCOM metric and give a new definition as the number of pairs of methods in a class having no common attribute references minus the number of pairs of methods that have common attribute references [CK94].

Hitz and Montazeri present two interpretations of the LCOM metric, referred to as LCOM3 and LCOM4 by Briand *et al.* [HM95, BDW98]. LCOM3 is defined as the number of connected components in a graph where the nodes in the graph are the methods of a class c and an edge exists between two methods if they both access at least one common attribute. They note that the presence of access methods in a class has the undesirable effect of artificially decreasing the cohesion of a class when measured using LCOM3 as methods may not directly access the attributes of a class but instead make use of the accessor methods. To address this problem they propose an alternative LCOM definition, LCOM4 which takes into account method invocations. In situations where there is only one connected component in the graph, the LCOM4 metric can be further assessed using the Connectivity (Co) metric by taking into account the number of edges of the connected component [HM95, BDW98].

Bieman and Kang identify a problem with the LCOM metric, noting it is effective at identifying very uncohesive classes, but it is not so effective at distinguishing

between partially cohesive classes. They propose two cohesion metrics, Tight Class Cohesion (TCC) and Loose Class Cohesion (LCC) that count the number of connected methods in a class. Like the LCOM metric, two methods are connected if they both access at least one common instance variable of the class, however a distinction is made between methods which access attributes directly or indirectly [BK95, BDW98].

Further criticism of the LCOM metric comes from Henderson-Sellers who identifies two major problems with the metric. First, there are several examples of dissimilar classes who all have a value of 0 for LCOM. Second, there are no guidelines for comparing LCOM values as the metric yields a different range of values for different classes. As a result, he proposes an improved version of LCOM, called LCOM* or LCOM5 by Briand *et al.* [BDW98], which considers the notion of "perfect cohesion". A situation where each method of a class references every attribute in the class yields a value of zero for the measure, a situation where each method of a class references only a single attribute yields a value of one and every other particular case can be represented as a percentage of this perfect value [HS96, BDW98].

Lee *et al.* propose a number of metrics that are based on the flow of information through a program [LLWW95]. They propose the metric (ICH), defined for a method `m` implemented in a class `c` as the number of invocations to other methods implemented in class `c`, weighted by the number of parameters of the invoked methods. The more parameters that an invoked method has, the more information is passed between the invoking and invoked method and therefore the stronger the connection is. It is possible to scale this metric to the class level by summing the the ICH values of each of the methods in a class and similarly to the system level by summing the ICH values of all the classes in the system [LLWW95, BDW98].

A set of cohesion measures for object-based systems are described by Briand *et al.* [BMB94] and adapted to object-oriented systems by Briand *et al.* [BDW98]. For this a class is considered as a set of data declarations and methods and different types of interactions are defined between the data declarations and methods. The metrics Ratio of Cohesive Interactions (RCI), Neutral Ratio of Cohesive Interactions (NRCI), Pessimistic Ratio of Cohesive Interactions (PRCI) and Optimistic Ratio of Cohesive Interactions (ORCI) are defined in terms of these different types of interactions. A complete formal definition is not given for these metrics as the definitions for the data declaration interactions are specified informally [BMB94, BDW98]. Therefore we omit these metric definitions in our work.

2.1.4 The Elusive Goal of Precise Metric Definitions

One way to establish that a software measure is meaningful and useful in practice is through empirical validation [BBM96]. For software metrics to be considered successful their empirical validation is of crucial importance and several attempts have been made to validate various different software metrics [CDC98, BWDP00]. However, one of the problems with software metrics is that they can be easy to define, but difficult to justify or correlate with external attributes [MP07]. One of the main reasons is the way software measures are defined and this has been noted by several authors in the literature [BDW98, BBA02, KHL01].

Briand *et al.* note that measures are often not accepted in the industry because the concepts they work with are not precisely defined [BMB96]. The problem of imprecise and ambiguous metric definitions have also been noted by several other researchers [BDW98, BDW99, BBA02, LLL08]. Kitchenham *et al.* assert that measures that are incompletely or poorly defined or documented results in invalid or incomparable data being collected [KHL01]. These problems hinder the empirical validation of software metrics and consequently their adoption by the software engineering community. Because of these problems research has been conducted to attempt to address these issues with metric definitions and we discuss some of this research in the remainder of this subsection.

Kitchenham *et al.* note the importance of validating software measures and outline a framework for software measurement validation. The framework is based on two models, a structure model and a definition model. The structure model represents the various elements involved in software measurement including the entities to be measured, their properties, measurement units and the relationships between these elements. The definition model represents how these structural elements are defined when specifying a software measure. They also define ways to theoretically and empirically validate software measures in terms of these models [KPF95].

According to Briand *et al.* the ad hoc manner in which object-oriented metrics are developed results in imprecise and ambiguous metric definitions which limit their use and adoption by the metrics community [BDW98, BDW99]. This allows for different interpretations of metric definitions by different researchers. It also makes it difficult to perform independent validation of empirical results relating to metrics and to investigate how different metrics relate to each other. In reviewing over 40 coupling and cohesion measures that exist in the literature they identify sev-

eral gaps in their specifications. For example, several of the measures considered by Briand *et al.* are based on methods of a class and for many of these metrics it is unclear what constitutes a method. For example, should constructors, finalisers/destructors and accessor methods be considered? Should methods that are inherited but not defined in a class be included? Should abstract methods count as empty methods, or not at all? To address this problem they first define a standard terminology and formalism that aims to ensure that metrics are defined consistently. Using this notation and formalism they present definitions for over 40 coupling and cohesion measures. Finally they present two unified frameworks, one for coupling measures and one for cohesion measures. The frameworks define what constitute coupling and cohesion and provide a classification for coupling and cohesion measures. The frameworks are developed in an effort to provide researchers with a way to compare different coupling and cohesion metrics and to determine the potential use of the metrics [BDW98, BDW99].

Although a proper terminology and formalism is important to facilitate unambiguous metric definitions, we believe that adoption of this formalism would be greatly aided by the ability to automate software measurement directly from the metric definitions expressed in this formalism. We use this terminology and formalism as a basis for developing our measurement metamodel, the results of which are discussed in Chapter 4.

More recently, researchers have turned to the domain of MDE using the concepts of models and metamodels for the definition and implementation of software metrics [MP07]. We first introduce the domain of MDE and its related concepts in the next section before discussing this work in more detail.

2.2 Model Driven Engineering

Model Driven Engineering (MDE) also referred to as Model Driven Development (MDD) is an emerging approach to software development in which models are the primary focus. The term was first introduced by Kent [Ken02] and is described as "simply the notion that we can construct a model of a system that we can then transform into the real thing" by Mellor *et al.* [MCF03]. The main idea of the approach is that a software system is specified at various levels of abstraction using different modelling languages and that this specification is iteratively transformed into a concrete model or implementation. There are currently a number of tools and

frameworks that support the MDE approach [Pla].

In general, MDE can be accomplished in several different ways using a number of different standards. One of the most well known realisations of MDE is the Model Driven Architecture (MDA ™) proposed by the Object Management Group (OMG) [OMG01a, Fav04, ABE+06]. The basic principle of MDA is to define a platform-independent model (PIM) and transform it to one or more platform-specific models (PSM) and then transform the PSMs to code. In MDA, the process of transforming between PIM and PSM and PSM and code is performed automatically. It is claimed that the main benefits of MDA are improvements to productivity, portability, interoperability and maintenance and documentation within the software development process [WKB03, ABE+06].

MDE and MDA rely on a set of common concepts such as model, modelling language, metamodel and model transformation. Moreover, specific to MDA are the standards and technologies used to implement these common concepts such as MOF, UML and QVT. As many of these concepts are core to our measurement approach we also discuss them in the remainder of this section.

2.2.1 Models and Metamodels

Models provide a representation of a real system and are increasingly important in software engineering, particularly in MDE [Sei03]. Typically, we think of a model of a software system as being a design model, such as UML class or sequence diagrams, or an implementation model, such as an actual program. For example, a simple University system may be represented using a UML class diagram as shown in Figure 2.1 (1). This class diagram contains classes such as `Student` and `Course` which represent the entities in the system and associations such as `enrols` which represent the relationships between these entities. The same University system can also be represented using a programming language such as Java or C#. Thus the same system can be modelled using different modelling languages.

Each of these different modelling languages (UML, Java, C# etc.) can also be described using a model which is referred to as a *meta*model. Essentially, a metamodel defines the terms and concepts of the modelling language. One example of a metamodel specification is the *UML Superstructure Specification* from the OMG [OMG07b], which defines the constructs that can be used in each of the diagrams of the UML. A simplified extract from this metamodel is shown in Figure 2.1 (2).

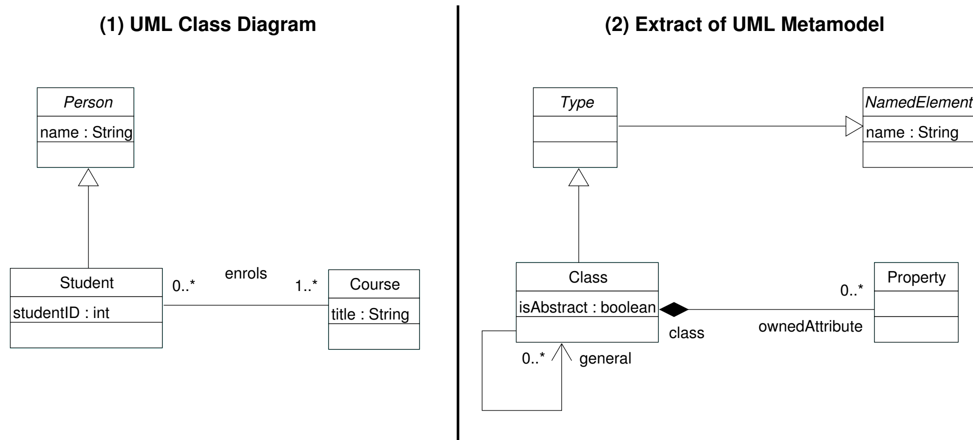

Figure 2.1: A sample UML class diagram and simplified extract from the UML metamodel. *This shows a class diagram representing a simplified University system along with a simplified extract of the UML metamodel [OMG07b] showing the parts relevant for modelling this class diagram.*

This figure depicts the various different constructs found in the class diagram in Figure 2.1 (1) such as `Class` and `Property` and the relationships between these entities. Finally, it is also possible to create a model that describes the constructs in the metamodel and such a model is called a *meta*metamodel and one such example is the Meta Object Facility (MOF) [OMG06a].

2.2.2 Model Transformations

Model transformations are yet another concept that play an important role in MDE. According to the OMG a model transformation is “the process of converting one model to another model of the same system” [OMG01a]. Model transformations are used for a variety of different purposes including model refactoring, model refinement and code generation from models [CH06].

Figure 2.2 presents an overview of the model transformation process. This figure shows that a model transformation takes a model as input, referred to as the source model and produces as output another model, referred to as the target model. Both the source and target models in a transformation must conform to their respective metamodels. A model transformation is composed of a set rules specified using a transformation language which conforms to its respective metamodel. By referring to the metamodels of the source and target models, a transformation definition specifies what or how elements in the source model are transformed to elements in

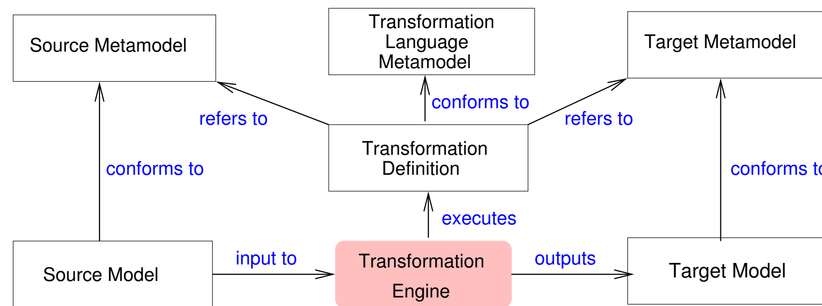

Figure 2.2: Overview of the model transformation process. *An overview of all the various different concepts involved in performing a model transformation.*

the target model. A transformation engine is then used to interpret the definition and perform the transformation on supplied models [WKB03, CH06].

In recent years a number of transformation languages and approaches have been proposed such as Kermeta and ATL [MFJ05, JK05]. Czarnecki and Helsen present a comprehensive list and description of the various different model transformation approaches available which include: relational/logic, graph-based, direct-manipulation, structure-driven and hybrid [CH06]. To facilitate the comparison of these different approaches and aid developers on their choice of transformation language they have proposed a framework for classifying the approaches which is based on a feature model [CH06]. Other classification efforts have involved the definition of a taxonomy of model transformations which includes classifying transformations as endogenous or exogenous and horizontal or vertical as well as outlining some important characteristics to consider when evaluating model transformation languages and tools [MG06].

2.2.3 Metamodelling Architectures

The relationship between models, metamodels and meta-metamodels is commonly depicted as a hierarchy that spans multiple levels; such a hierarchy is referred to as a metamodelling hierarchy or metamodelling architecture. The hierarchy can be viewed from two perspectives, one is that each model is an instance of the model that is directly above it in the hierarchy or alternatively each model is a model of the model that is directly below it in the hierarchy. This structure can be applied recursively yielding the possibility of an infinite number of meta-layers [OMG07a]. Some examples of metamodelling hierarchies include the Resource Description Framework (RDF) and the Extensible Markup Language (XML). RDF

	Description	Examples
M3	Meta-metamodel layer	MOF [OMG06a] entities, e.g. Class, Property, Operation
M2	Metamodel Layer	UML metamodel [OMG07b] describing the entities in a UML model, e.g. Package, Type, Class, Operation
M1	Model Layer	A UML model, e.g. class diagram, sequence diagram
M0	Data Layer	The run-time instances of the elements in the UML model, e.g. run-time objects

Table 2.1: The Four Layer Metamodel Architecture. *This table shows the standard four-layer hierarchy, using the UML modelling as an example (p19 of [OMG07a]).*

was defined by the World Wide Web Consortium (W3C) to support the representation and exchange of information on the web [W3Cb]. XML also defined by the W3C is a framework for defining customised markup languages [W3Ca].

Traditionally, modelling within the OMG is based on a four-layer hierarchy [WKB03] which is described in Table 2.1. The most abstract layer, *M3*, is the layer that contains the meta-metamodel or the formalism used to define modelling languages. A number of different formalisms or languages can be used to describe metamodels and in MDA this is the MOF. Beneath this is the *M2* layer which consists of the metamodel for the modelling language, for example the UML metamodel. The *M1* layer represents a model of a real system such as a class diagram of a UML model containing entities such as classes and attributes. The *M0* layer represents the actual run-time instances of the model elements defined in the model at layer *M2*.

Where possible our measurement approach makes use of the OMG standards for MDE and we outline these standards here.

MOF

The Meta Object Facility (MOF) is the OMG's standard formalism for constructing metamodels or defining modelling languages in MDA [OMG06a]. It provides a set of constructs for describing metamodels including `Class`, `Property` and `Operation`. MOF is reflective which means it is used to define itself and therefore in terms of the OMG 's metamodelling hierarchy it is at the top layer with no need to have extra meta-layers above it. The MOF is described by an OMG specification

document which also contains a comprehensive list of requirements that must be met for a metamodel to conform to the MOF ([OMG06a], p. 35, 48). For example, there is no support for n-ary associations in the MOF and therefore all associations in a metamodel must be binary. Also, the MOF does not support association classes and consequently they cannot appear in a MOF-compliant metamodel.

As well as facilitating comprehension, using a standard metamodelling formalism aides interoperability, through formats such as the XML Metadata Interchange (XMI) and also facilitates writing and implementing transformations from one modelling language to another.

UML

In terms of MDE, the UML is not a vital component but in terms of the MDA it is the proposed language for creating the models (PIMs and PSMs) of the system to be built [WKB03]. The UML was created as a language for specifying, visualising, constructing and documenting the artifacts of software systems as a result of the integration of the concepts of Booch, Jacobson and Rumbaugh. In 1997, the UML was adopted by the OMG as a standard for modelling object-oriented systems and has since become the de-facto modelling language [BRJ99]. Since then, the UML has undergone several revisions from versions 1.4, 1.5 up to the current version 2.1. The UML 2 saw the introduction of several new diagrams and now contains 13 diagrams that can be used to present different views of an object-oriented system [OMG01b, OMG03, OMG07c].

The OMG have written specification documents defining the UML in terms of its concrete syntax, abstract syntax and semantics. The concrete syntax is a graphical notation, the abstract syntax is described using a set of UML class diagrams and the semantics are described using a set of well-formedness rules written using natural language and the OCL [OMG07a, OMG07c].

OCL

The Object Constraint Language (OCL) is a standard language that is used to write expressions about elements in object-oriented models in a clear and unambiguous manner. It is used to add information to object-oriented models that cannot be expressed by using UML diagrams alone. It also offers the ability to navigate over instances of object-oriented models, allowing for the collection of information about

the navigated model [WK03].

The OCL was initially described as part of the UML 1.1 specification and was limited to writing constraints on UML models [WK03]. Realising that it should be possible to express far more additional information in a model than just constraints the OMG began working on a new version of OCL, OCL 2.0. This new version of OCL has been defined in its own specification and adopted as an OMG standard [OMG06b].

There are several differences between OCL 1.1 and 2.0. As well as being a constraint language, OCL 2.0 is now also a general query language. The language has also been extended with many new features, including the introduction of the `context` keyword for specifying the element to which the OCL expression is to be attached and the introduction of new types such as `Tuple`, `TupleType` and `Undefined Value` [WK03]. These new aspects of OCL 2.0 have made it possible to use OCL to define certain metrics which previously were very difficult or almost impossible to do.

XMI

MDA uses the XML Metadata Interchange (XMI) standard for representing and exchanging MOF models via XML [OMG07d]. A MOF model is any model whose metamodel is defined in terms of MOF. Essentially, the MOF standard defines what meta-data information about the model is to be stored and how it is to be organised, ensuring that all models are represented consistently in XML format. A common application of XMI is as a model interchange format for the UML, thus facilitating the exchange of UML models between different UML modelling tools. At present many of the UML modelling tools do not support the full XMI standard or alternatively implement their own variant of the standard. As a result there are a number of inconsistencies and incompatibilities between these tools which makes the exchange of models between the tools very difficult and in many cases impossible.

QVT

In MDA, Queries/Views/Transformations (QVT) has been adopted as the standard for specifying model transformations [OMG08b]. In 2002, recognising model transformations as an integral part of the MDA paradigm, the OMG issued a Request For Proposal (RFP) for a standard language for writing model transformations for

MOF models. The specification outlined several requirements for the language such that the transformation definition language must be declarative and that the abstract syntax of the language must be defined as a MOF metamodel [OMG02]. Several proposals were submitted in response to this RFP and in 2005 the OMG released a draft proposal for the QVT language which is currently at version 1.0 and has been adopted as an OMG standard [OMG05a, OMG08b].

QVT adopts a hybrid approach to model transformations, composed of a mix of declarative and imperative languages. QVT also incorporates the OCL 2.0 and extends it to an imperative form of OCL. Several transformation languages and tools claim to be QVT-compliant. However, due to the infancy of the standard many of these tools are still in the early stages of development and are not yet fully robust or reliable.

2.2.4 Eclipse Modelling Framework

The Eclipse Platform is a free and open source integrated development environment (IDE). It has been designed to be extendible and provides a plug-in architecture which enables software developers to create tools that integrate with, and build upon the functionality of other tools, including the Eclipse platform itself [Eclb]. The Eclipse Modelling Framework (EMF) is a framework that supports the creation of tools and applications using MDE principles. Initially, the main goal of EMF was to provide an implementation of MOF for the Eclipse platform but over time it evolved into a modelling and code generation framework. EMF provides a metamodel called Ecore, which is very similar to a core subset of MOF and is used to create simple data models. These data models are created either directly in Ecore, from annotated Java, XML documents or modelling tools such as Rational Rose [Eclc, BSM+04]. EMF offers the ability to generate Java classes for the elements in these models as well as Java classes which can be used for viewing and editing the models and a reflective API for manipulating EMF objects directly. It also provides persistence support for Ecore models and model instances in XML format thus facilitating interoperability with other tools and applications. By providing a working implementation of the MOF, EMF has paved the way for the implementation of other OMG standards. For example, the UML2 Eclipse project has been created which provides a usable EMF-based implementation of the UML 2 metamodel for the Eclipse platform [Ecle]. This project has proven extremely useful as it has been

used as a basis for developing modelling tools such as Papyrus [CEA], used for the interchange of UML models between tools and it has also facilitated the testing and validation of the OMG's specification of the UML metamodel [Ecle].

2.3 Metamodelling and Software Metrics

Several authors have attempted to address the problem of ambiguous metric definitions. Many of these approaches involve modelling the entities to be measured, and then defining the metrics in terms of this model. In standard terminology, the metrics are defined on the metamodel of the entities being measured [MP07]. Such an example, mentioned earlier in Section 2.1.4, is the canonical presentation of coupling and cohesion metrics by Briand *et al.* which was effectively based around a metrics specific metamodel of an object-oriented software system [BDW98, BDW99].

Based on a review of the state of the art of object-oriented software design metrics, Abounader and Lamb propose a data model of design information for software metrics with the intention of implementing it in the form of a database [AL97]. They believe that such a database combined with adequate tool support for extracting the design information from a software design or implementation would benefit the software metrics community by making it easier to compare and validate large numbers of software metrics. They provide a summary of the entities and relationships in their data model but do not provide details about the database implementation or how to extract the information from a software design or implementation for storage in the database [AL97].

Another solution put forward by Reißing involves the proposal of a formal model on which to base definitions of object-oriented design metrics [Rei01]. This model is called ODEM (Object-oriented DEsign Model) and consists of an abstraction layer built upon the UML metamodel. However, this model can only be used for the definition of metrics for UML and does not solve the ambiguity problem as the abstraction layer consists of natural language expressions.

Mens and Lanza propose a language independent metamodel for object-oriented metrics that is based on graphs [ML02]. They use this to define a selection of generic object-oriented metrics and higher order metrics but do not consider coupling or cohesion metrics.

As part of the European Esprit Project FAMOOS, one initiative has been to

develop approaches and tools for re-engineering large scale object-oriented software systems written in different languages such as C++, Java, Smalltalk and Ada [LD02]. This project has resulted in the creation of the Moose Re-engineering Environment which is based on a language independent metamodel called FAMIX (FAMOOS Information eXchange model) [DTD01]. Moose works by extracting the relevant information from source code and mapping it to the language independent representation, FAMIX. Included in the Moose Re-engineering Environment is a metrics tool that uses the language independent metamodel representation of the software to compute the metrics. The metrics engine computes more than 50 different metrics of which about 30 are language-independent. However, many of these metrics are primarily counting metrics and there is no support for coupling and cohesion metrics which the authors say is because of the lack of consensus on how to define many of these metrics [LD02]. Furthermore, there is no information on how the approach can be applied at the design level (e.g. UML models).

Using a clearly defined metamodel is important for facilitating unambiguous definitions of metrics, but it also has clear advantages in terms of implementation. Many metamodelling frameworks facilitate the implementation of corresponding APIs that allow for the representation and traversal of model instances, for example MOF and EMF [OMG06a, BSM$^+$04]. Previous research has exploited this implementation aspect of metamodels by defining metrics as queries over metamodels or metamodel based repositories.

Wilkie and Harmer develop an extensible metrics analyser tool for object-oriented programming languages [WH02, HW02]. The tool is based on a general object-oriented programming language metamodel in the form of a relational database schema. Metric definitions are expressed as SQL queries over this schema. The tool is extensible as it has support for incorporating new metrics and new object-oriented programming languages. However, the complexity of the approach is similar to employing a programming language to define and implement the metrics as it requires the additional effort of developing C code to execute the SQL queries. A very similar approach to Wilkie and Harmer is that of Scotto *et al.* who also propose a software metrics tool, called *WebMetrics* which calculates metrics by evaluating SQL queries on a relational database [SSSV04]. They claim the main advantage of their tool is that it separates the parsing of the source code from the computation of the metrics. This approach suffers similar drawbacks as that of Wilkie and Harmer as it also involves the complexity of developing source code for the SQL queries.

Both approaches are specific to source code and do not take into consideration metrics applied at higher level of abstractions such as UML-based metrics.

The use of the OCL as a way to define software metrics was first proposed by Abreu and expanded by Baroni *et al.* who propose using the OCL and the UML metamodel as a mechanism for defining UML-based metrics [Abr01, Bar02, BA02]. They have used the approach to define the CK metrics [BA03b] and have built a library called FLAME (Formal Library for Aiding Metrics Extraction) which is a library of metric definitions formulated as OCL expressions over the UML 1.3 metamodel [Bar02, BA03a, OMG00]. Goulão *et al.* have also employed this approach for defining component based metrics and used the UML 2.0 metamodel as a basis for their definitions[GaA04, OMG05b].

El-Wakil *et al.* propose the use of XQuery as a metric definition language [EWEBRF05]. They propose extracting metric data from XMI design documents, specifically UML designs. XQuery is a language that can be used to query and extract information from XMI documents. Again this approach has only been used to define metrics at the design level, specifically for UML designs. There is no information available on how it extends to other languages.

Marinescu *et al.* propose a simplified implementation of object-oriented design metrics using a metric specification language called SAIL [MMG05]. The language is built on top of the MEMORIA metamodel. The disadvantages of this approach are that the langauge and metamodel used is non-standard and the issue of representing source languages using the MEMORIA metamodel is not addressed. Furthermore, there is no mention of how the approach can be used to automate the measurement of software.

Other related work involves the development of tool support for measuring object-oriented metrics using a metamodel or repository type approach. These include a UML measurement tool by Lavazza and Agostini and the SDMetrics measurement tool [LA05, SDM06]. Both these tools are extendible in that they have support for user-defined software metrics. However they are limited to measuring metrics from UML designs.

The majority of the work in the literature to date has concentrated on using metamodels on their own or exploiting the implementation aspect of metamodels for software measurement. However, very little attention has been given to investigating how to apply an entire MDE approach to software measurement in order to address the problems outlined earlier in this Chapter. Very recently Monperrus

et al. propose a model-driven approach to software measurement referred to as the MDM (Model-Driven Measurement) approach which involves specifying metrics as models which are instances of a metric specification metamodel [MJCH08]. The MDM approach differs from ours in that our approach involves modelling the entities to be measured and expressing the metrics as queries over these entities whereas this approach involves modelling the metric definitions themselves and transforming other domain models to the metrics specification metamodel in order to evaluate the metrics. The MDM approach has clear advantages including being domain independent, being applicable to any modelling language and facilitating the automatic generation of measurement tools from the metric specifications. However, this requires the user to learn a new language to express the metrics. With the multitude of languages that already exist, developing and learning a new language adds an extra, unnecessary level of complexity. Furthermore, the type and number of existing metrics that can be implemented using the MDM approach is limited by the expressive power of the language. It is not possible to use the MDM approach to implement certain cohesion metrics as the concept of method pairs can not be represented using the language and the language needs further extension to incorporate this feature. We believe it may be worth investigating combining this approach with our OCL-based approach as noted by the authors of the MDM approach [MJCH08]. Another disadvantage is that when new metrics are proposed the MDM approach requires all other languages to be transformed to this metric requiring several new transformations.

A related initiative is the RFP for a software metrics metamodel issued by the OMG Architecture Driven Modernisation Task Force [OMG06c]. The objective of this is to define a framework for the representation and exchange of information related to software measurement. The RFP is not very exact about what this metamodel should represent but it does aim to encompass information broader than just the specification of metric definitions. Recently, a proposal for a software metrics metamodel has been proposed in response to this RFP [OMG08a]. This proposal attempts to address the representation of three different aspects; the measurement process, the specification of software measures and the representation of the measurement results [MJCH08]. These proposals exist only as specifications and there is no information on how to implement the specifications or how well they work in practice. Our work is concerned only with the specification of software measures.

An important aspect of any approach is that it is reliable and correctly computes

the software metrics, to ensure this requires adequate support for validating the approach. However, none of the approaches outlined above attempt to tackle this issue. Furthermore, this issue becomes even more important in an MDE context as the applications for measuring software are created automatically. One approach to validation is software testing and we develop testing techniques that can be applied within our MDE-based measurement approach. In the next section we review the relevant research related to software testing in MDE.

2.4 Software Testing in Model Driven Engineering

Software testing is an important and integral part of the software development process. It is used to reveal bugs in a system, to assure that the system complies with its specification and to verify that the system behaves in the intended way. Various definitions have been presented for software testing [Bei90, Bin00]. For example, Myers [Mye04] defines it as:

> *"... the process of executing a program [or system] with the intent of finding errors."*

For our purposes, we will consider software testing as the process of determining if the observed behaviour of a system corresponds with the expected behaviour of the system. This process involves executing the system on a set of inputs and determining if the actual behaviour of the system corresponds to the expected behaviour. These inputs to the system are known as test cases and a collection of test cases is referred to as a test set or a test suite. With this notion of testing, a mechanism referred to as the test oracle is used to determine whether or not the results of the test execution are correct. Commonly, this is achieved by comparing the output produced by the execution, either manually or automatically with the pre-computed expected output [Wey84].

Software testing in the context of MDE is very similar, as test cases are created and selected for testing and a test oracle is defined to check the correctness of the system after it has been executed with the selected test cases. However, there are a number of limitations to using existing testing techniques in the domain of MDE, in particular for testing model transformations. The main entities involved in MDE are models which are inherently complex and generating test data using traditional techniques is awkward and inefficient [FSB04, BDTM+06]. As a result, research

has been conducted to develop and adapt existing techniques for test case generation to MDE.

2.4.1 Test Case Generation

Gogolla *et al.* describe an approach to the automatic generation of model instances (snapshots) from UML class diagrams [GBR05]. ASSL (A Snapshot Sequence Language) is used to specify properties of a required model instance. Using their approach they generate two types of model instances, those that are test cases and those that are validation cases. The test cases confirm that models with certain properties can be created from the specification. The validation cases are used to show that certain properties of a model are a consequence of existing properties of the model. However, this approach is not fully automated as it requires the creation of scripts for each model in order to generate model instances.

An approach to metamodel instance generation is presented by Ehrig *et al.* [EKTW06]. This approach involves the automatic creation of an instance-generating graph grammar for the given metamodel. They also describe how to translate restricted OCL constraints to graph constraints. The grammar and the graph constraints are then used to create metamodel instances. However this approach does not support attribute values, only supports limited OCL constraints and cannot be used to verify properties of the metamodel.

Based on their experiences of designing and implementing model transformations for business process models, Kuster *et al.* present a discussion on model transformation testing [KAER06]. Their model transformations are specified initially as a set of abstract rules that are not executable and are then iteratively refined and eventually implemented directly in Java. Based on this they describe three white-box testing techniques for the construction of test cases for model transformations. The first approach, referred to as the model coverage technique, uses a specially developed template language and involves converting each of the abstract transformation rules into a metamodel template. Several instances of the template are generated automatically, thus creating a set of test cases for the transformation rule from which the template is derived. The second approach is based on identifying the metamodel elements that are transformed by the model transformation and then selecting those elements that are associated with constraints in the target metamodel. Then for each of these constraints, a test case is generated that aims to

validate that the constraint is not violated under the transformation. The third approach uses the concept of rule pairs to generate test cases. The main idea is that test cases are generated by pairing the transformation rules. Based on each rule pair, all the possible overlaps of the model elements referred to by the rules are calculated. The two models are combined based on their overlapping elements to produce test models.

Lamari address the challenges of automatic test case generation by adopting traditional functional techniques [Lam07]. A special purpose formal language for specifying model transformations is proposed. The purpose of the language is to facilitate the automatic parsing of specifications which in turn supports the automatic generation of test cases. In addition, a method is described for the structural decomposition of a metamodel which when combined with the traditional functional testing techniques of category partitioning and the classification tree method can automatically generate test cases from a metamodel [Lam07].

Fleurey *et al.* also adapt existing functional testing techniques for the automatic generation of test cases in the domain of model transformation testing. They refer to such test cases as test models. This technique is based on a systematic algorithm that iteratively generates test models from an effective metamodel, where the effective metamodel is the part of the source metamodel that is relevant to the model transformation under test. The algorithm makes use of a set of test adequacy criteria for generating the test models [FSB04].

2.4.2 Test Adequacy Criteria

One important aspect of software testing is deciding when enough testing has been done. How do you decide if a set of tests are adequate? This question was first addressed by Goodenough and Gerhart [GG75] when they considered the idea of a test adequacy criterion, that is, a criterion that defines what makes an adequate test. A test adequacy criterion is a rule or a set of rules that impose requirements on a test set [ZHM97]. Adequacy criteria play an important role in the testing process. They can be used as a stopping rule. Testing stops when enough test cases have been produced to satisfy the criteria. They can also be used as a measurement of test quality. A measure of adequacy is associated with a test set, therefore different test sets can be compared in terms of their adequacy measurement. Adequacy criteria also provide a basis for generating test cases, that is test cases are generated (often

automatically) to meet the adequacy criteria. Coverage can be used to measure the extent to which an adequacy criterion is satisfied. The percentage of requirements (as specified by the adequacy criterion) that are satisfied by a test set can be used as an adequacy measurement. Therefore coverage criteria are a type of adequacy criteria that specify the percentage of requirements that must be covered [ZHM97].

For conventional programming languages the degree of coverage of elements such as statements, branches, paths, functions etc. can be calculated for a test suite, with the goal of achieving 100% coverage of the chosen element [Bei90]. Given the widespread use and acceptance of such measures in the programming domain, it is natural to consider their use for modelling and model transformation.

A range of coverage criteria have been suggested for the various UML diagrams [MP05]. For example, Andrews *et al.* define a number of coverage measures for class diagrams consisting of the association-end multiplicity (AEM) criterion, the generalisation (GN) criterion and the class attribute (CA) criterion [AFGC03]. The AEM criterion requires that for each association in the class diagram a set of representative multiplicity pairs are created. The set of multiplicity pairs that are to be covered are derived from the class diagram using a modified form of category-partition testing [OB88]. This involves the partitioning of the value domain of the multiplicity into equivalence classes and the selection of a single value from each class. For each end of the association a set of possible multiplicity values are selected in this way. Each value from the first set is combined with each of the values from the second set, thus producing a set of multiplicity pairs. The GN criterion requires that testing cause each generalisation/specialisation relationship (or specialisation element) of a class diagram to be created at least once. The CA criterion requires coverage of a set of attribute value combinations for each class in the class diagram. The category partition method is used to produce a set of possible values for each attribute in a class. Elements from each of these sets are combined to create a set of attribute values for each class [AFGC03].

Since a MOF metamodel can be described using a UML class diagram, coverage criteria for class diagrams provide a basis for developing similar criteria for metamodels. This is the approach taken by Fleurey *et al.* in their approach to model transformation testing [FSB04]. Their approach generates test models from an effective metamodel using a set of test adequacy criteria. The criteria they use are the CA and AEM criteria of Andrews *et al.*. They omit the GN criterion, reasoning that their approach is tailored to testing model transformations in which the emphasis

is on the structure of the model rather than behaviour. For the two chosen criteria, a set of representative values for each of the elements in the effective metamodel is created. All valid combinations of these values are then determined and used to create test cases or test models. However, OCL constraints, an important part of a metamodel, cannot be directly reflected, leading to an under-specification of model instances.

Furthermore, the work of Fleurey *et al.* relies on generating a test suite that adequately covers the input domain of a transformation, as defined by the input metamodel, or a relevant subset. However, there is little work on directly considering the coverage of the transformations only. One related area is that of grammar testing, since the process of transforming an input language using a grammar (or a generated parser) is analogous to a model transformation. Various coverage criteria have been proposed for grammar testing, the most simple being rule coverage, which requires that each rule in the grammar be used during testing, although there are many more complex variations [Läm01, LS06].

A model transformation consists of more than just rules to match the input, and so any consideration of coverage should also deal with model generation and any internal operations. To date there has been relatively little work on linking coverage of the "front end", as defined by a grammar or the input metamodel, with coverage of the "back end" as defined by transformation internals and generation code. Hennessy and Power show that applying test suite reduction using only grammar coverage as a criteria yielded poor results for the internals of a C++ parser [HP08], and thus would suggest that coverage of transformation internals should also be considered. We explore this idea further in Chapter 5.

Brottier *et al.* build on the work of Fleurey *et al.* and create a test generation tool that uses the approach outlined by Fleurey *et al.* [FSB04] to generate test cases [BFS+06]. The tool generates test models from a metamodel and a set of model fragments only. The model fragments must be defined by the tester manually. In addition, the test generation process does not take the model transformation into consideration and as a result it is not apparent how to automatically generate the effective metamodel or model fragments. To address these issues Wang *et al.* extend the work of Fleurey *et al.* and Brottier *et al.* by developing a prototype tool that generates test models automatically [WKC08]. The tool automatically generates an effective metamodel from a model transformation and uses all three criteria of Andrews *et al.* to create coverage items for the effective metamodel and then gen-

erates test models to satisfy these coverage items. The tool is limited to the Tefkat transformation language. Transformations are checked by manually comparing the actual outputs of the transformations with the expected ones [WKC08]. This approach is tedious and error-prone and does not scale well to very large number of test cases. This limitation has also been noted by the authors and as future work they intend to investigate approaches for automatically generating test oracles.

2.4.3 Test Oracle Construction

There has been very little research in the literature that addresses the problem of creating test oracles for model transformations. Some research that has briefly touched this issue is that of Lin *et al.* who propose a testing framework for model transformations [LZG05]. They identify three challenges for model transformation testing; automatic comparison of models, visualisation of model differences and debugging of the model transformation definitions. In the context of their framework, they address the first two of these challenges by exploiting model comparison techniques to provide tool support for test case construction, test case execution and comparison of test case results with expected results. However, they fail to identify the challenge of automatic test case generation or appropriate test adequacy criteria and test cases are created manually in their framework [LZG05].

Baudry *et al.* discuss a possible approach to test oracle development in the context of model transformation testing and outline the possible limitations of such an approach. This approach is based on the notion of contracts which specify constraints such as pre- and post-conditions on various elements of the transformation such as the transformation specification and output models produced by the transformation. They conclude that developing a general purpose solution for developing test oracles for model transformations is a very difficult task to achieve and propose that a more appropriate solution may be to group model transformations into different categories and develop different oracle development techniques for these categories [BDTM$^+$06].

2.5 Summary

The goal of software measurement is to provide a quantitative measure of a particular attribute or characteristic of a software system such as size or complexity and

to correlate this measure with some external attribute such as software quality or development cost. Many software metrics have been proposed in the literature for this purpose. However, many of these metrics have not seen widespread adoption in industry due to the problems mentioned in Section 2.1.4 such as vague and imprecise metric definitions and inability to perform independent validation of empirical results for these metrics. To date, several authors have attempted to address these problems using metamodels or repository based approaches to software measurement. However, only a limited amount of work has been carried out to encompass the full MDE process into an approach to software measurement and none of the approaches consider the issues of validity or correctness. The work in this thesis attempts to address this by investigating how metamodels and MDE can be used to reliably automate the measurement of object-oriented software. Furthermore, we adopt techniques from the software testing domain to address the problem of assessing the correctness and reliability of our MDE-based measurement approach.

Chapter 3

Towards an MDE Approach to Software Measurement

In this chapter an approach to defining software metrics is presented. A prototype tool called dMML (defining Metrics at the Meta Level) has been developed to support this approach and details of this tool are discussed. An outline of how to apply the approach to both the UML and Java is also presented. Details of the approach described in this chapter and its applicability to the UML and Java have been published in McQuillan and Power [MP06d, MP06b].

3.1 Introduction

In Chapter 2 we discussed the extensive availability of software metrics and how many software metrics have been proposed and new metrics continue to appear in the literature regularly [FP96]. Many of the metrics proposed are incomplete, ambiguous and open to a variety of different interpretations [BDW98]. This makes it difficult to create general metric tools since many of the metrics proposed can be interpreted in several different ways and every time a new metric is proposed the metric tools need to be updated with this metric [ML02]. This in turn makes it difficult to perform independent validation of empirical studies related to software metrics and to investigate how different metrics relate to each other.

Like Briand *et al.* we believe that addressing these problems requires an approach to software measurement that uses a standard terminology and formalism

for expressing software metrics [BDW99, BDW98]. Having a standard terminology and formalism allows metrics to be expressed in a clear and precise manner thus supporting the comparison, evaluation and validation of existing software metrics as well as the proposal of new software metrics. However, we take the work of Briand *et al.* a step further since we believe that adoption of this formalism would be greatly aided by the ability to automate software measurement directly from the metric definitions expressed in this formalism. This not only encourages the adoption of such standards but can also significantly reduce the time, effort and cost of the software measurement process.

To address the problem of ambiguous and imprecise metric definitions Baroni *et al.* propose the use of the OCL and the UML 1.3 metamodel as a way to define design metrics, effectively proposing them as a terminology and formalism for expressing software metrics [Bar02, BBA02]. In our opinion this approach provides a useful mechanism for the clear and precise definition of software metrics as it provides several advantages. In particular, since the language or formalism proposed to define the metrics is OCL, which is is an OMG standard, it is familiar to software engineers and modellers and there are a number of tools that support the use of the OCL.

However, we have also identified a number of limitations of the approach of Baroni *et al.*, such as non-conformance to current OMG standards and lack of available tool support, which we discuss further in Section 3.2. The main goal of this chapter is to build on the work of Baroni *et al.* to overcome these limitations. To achieve this we aim to

- adapt the metric definition approach to make it reusable for other languages and metamodels, specifically we are concerned with MOF metamodels or languages that can be described using MOF metamodels.
- ensure the approach makes full use of the OCL 2.0 syntax in order to adhere fully to current OMG standards.
- develop a flexible environment that can be used to define software metrics and fully automate the generation of a measurement tool directly from the definitions.
- demonstrate the feasibility and robustness of the approach by using it to define and calculate metrics for "real world" programs.

```
Classifier::IAN() : Integer
= allInheritedAttributes()->size()
```

Figure 3.1: IAN Metric Definition. *This OCL code presents the* IAN *metric as defined by Baroni [Bar02]. It is part of a larger set of measures called FLAME and specifies that the metric computes the number of inherited attributes for a* `Classifier` *element of the UML 1.3 metamodel. Any auxiliary operations used in this definition can be found in [Bar02].*

3.2 Defining Metrics at the Meta-Level

In this section we give details of an approach for specifying software metrics that is based on the use of metamodels and the OCL. The use of OCL as a way to express design metrics was first proposed by Abreu [Abr01]. This approach is explored further by Baroni *et al.* by adapting it to define design metrics as OCL constraints over the UML 1.3 metamodel [OMG00]. This approach of Baroni *et al.* involves modifying the UML 1.3 metamodel by creating the metrics as additional operations in the metamodel and expressing them as OCL conditions [Bar02, BBA02]. The approach was used to define a library of measures called FLAME (Formal Library for Aiding Metric Extraction) [Bar02, BA03a]. This library is composed of a set of approximately 90 auxiliary functions that compute some basic metric values.

For example, the `allOperations` measure is defined by adding this operation to the `Classifier` element of the UML metamodel and expressing in OCL that this operation returns a set containing all operations of the `Classifier` including the inherited operations. Similarly, the *IAN* metric is defined by adding it to the `Classifier` element and specifying in OCL that this returns the number of inherited attributes of the `Classifier`. Figure 3.1 reproduces this *IAN* measure as an example of the format of the metrics defined by Baroni [Bar02]. The functions in FLAME are used as a basis for the formal definitions of four different sets of metrics totaling approximately 77 metrics [Bar02]. Any metrics that require knowledge of code internals, such as LCOM and MPC are not specified formally.

Although this approach has many advantages, we have identified a number of limitations of the approach. These are

- **Direct modification of the metamodel elements:** Modifying the metamodel elements directly is not an ideal approach to defining the metrics for several reasons. As more and more metrics are defined for a metamodel element,

the number of operations defined for that element also grows. This raises the possibility of making the metamodel cumbersome and difficult to understand as you cannot separate the operations relevant to the metrics from elements relevant to the metamodel. Also, a standard metamodel is necessary for modelling tools to allow interoperability between them. If the metamodel is constantly changing with the addition of metrics, it is almost impossible to expect the modelling tools to constantly update their version of the metamodel to reflect the new metrics. Furthermore, if a change is made to the metamodel that renders the metrics specification incorrect then these metrics must also be updated immediately in the specification for the metamodel specification to be correct. Finally, it is unclear where common auxiliary functions such as summing a list of numbers or finding the maximum number in a set of numbers should be defined.

- **No available tool support:** Although a metrics extraction framework is outlined by Baroni *et al.* [BGaA02, Bar02], it is not available for download or use. This framework uses the USE tool to define and evaluate the software metrics [GBR05].

- **Non-conformance to OMG standards:** The syntax used in the metric definitions is specific to the USE tool and is based on an early version of the OCL syntax, namely version 1.1 which was part of the UML 1.4 standard. Since then a new improved version of the OCL syntax has been proposed and adopted by the OMG. The approach should be supported by the correct use of the OCL syntax and any definitions should be written using this syntax, thus making the definitions standard and available for use with other modelling and OCL tools.

- **Limited application and demonstration of the feasibility of the approach:** The approach has not yet been applied to real world applications and has only been used to calculate the metrics for a single UML class diagram, the Royal and Loyal example of [WK03]. The resulting metric values are presented by Baroni [Bar02], however no details on the correctness of these values are supplied. There is no discussion on how the metric values produced were assessed in order to ensure that they are correct for the class diagram. It is vital that any such results be verified in order to demonstrate the reliability of the approach used to calculate them.

- **No automated support for creation of metamodel instances:** The USE tool requires that the users manually create instances of the metamodel. For the Royal and Loyal example Baroni has created a metamodel instance generator to instantiate the classes corresponding the meta-classes of the UML metamodel. It appears that this consists of a script that has been manually created and loaded by the user into the USE tool and used to create these instances. This script is not generalisable and is specific to the Royal and Loyal example. Therefore, any other UML models will also require their own script in order to instantiate the UML metamodel and evaluate the metrics. This is noted by Baroni who state that "one workload generator tool would be of great help because, frequently, the UML model instances are done by hand" [Bar02].

To address these issues we have extended the approach of Baroni *et al.* in a manner specifically designed to be re-usable for other MOF metamodels or languages that can be described using a MOF metamodel. The extension involves decoupling the metric definitions from the language metamodel by extending the metamodel with a separate metrics package as depicted in Figure 3.2. This figure shows the language metamodel that describes the domain for which the metrics are defined on the right and the `Metrics` package on the left. The dashed arrow indicates a dependency between elements in the Metrics package and elements in the language metamodel. The `Metrics` package contains a single abstract class `Metrics`, any auxiliary operations that are common to all sets of metrics are defined in this class.

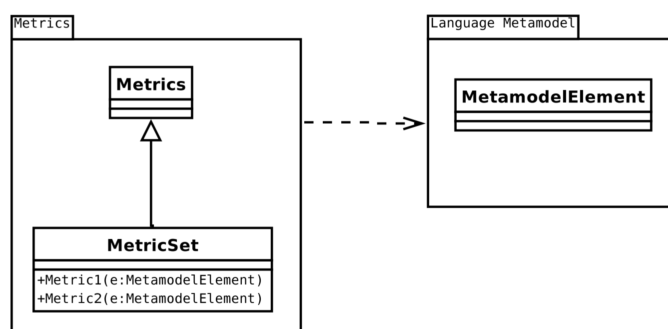

Figure 3.2: Extension to the language metamodel. *This package diagram shows the definition of a set of metrics as a separate package, with a dependency on (meta)classes from the language metamodel.*

Defining a group or set of metrics is a three step process:

1. A class is created in the `Metrics` package corresponding to the metric set; any auxiliary operations that are specific to that metric set are defined in this class. This is shown as a class called `MetricSet` in Figure 3.2.
2. For each metric in the set, a query operation in the metric set class is declared, parameterised by the appropriate elements from the language metamodel. In Figure 3.2 this is represented as the operations `Metric1` and `Metric2` both parameterised by `MetamodelElement` from the language metamodel.
3. The metrics are defined by expressing them as OCL query operations using the OCL `body` expression. The OCL expression specifies what attributes and associations of the metamodel are to be traversed to compute the result for the metric.

We have developed an easily extendible tool called dMML (defining Metrics at the Meta Level) that supports this approach and that can, in theory, be applied to any metamodel or language. It provides an environment within which users can specify metrics using the OCL 2.0 syntax and any MOF metamodel and automatically creates a measurement tool from these definitions. We have successfully used this approach to define and implement a set of metrics for both the UML and Java and applied them to a set of real world programs.

3.3 dMML: Tool Support for Defining Metrics at the Meta-Level

Developing tool support for the measurement approach requires the ability to evaluate OCL queries and expressions over MOF metamodels. At the time of implementation, we were unable to find adequate tool support for applying OCL to metamodels. This has not changed much which has been noted by Berkenkötter and Gogolla who observe that “Currently, only few tools are able to check OCL constraints on the model level let alone on the metamodel level” [Ber08]. Therefore, from an implementation point of view we choose to bring the metamodel down to the model level and treat the metamodel as a UML class diagram and the language instance/metamodel instance as instances of the class diagram or object diagrams.

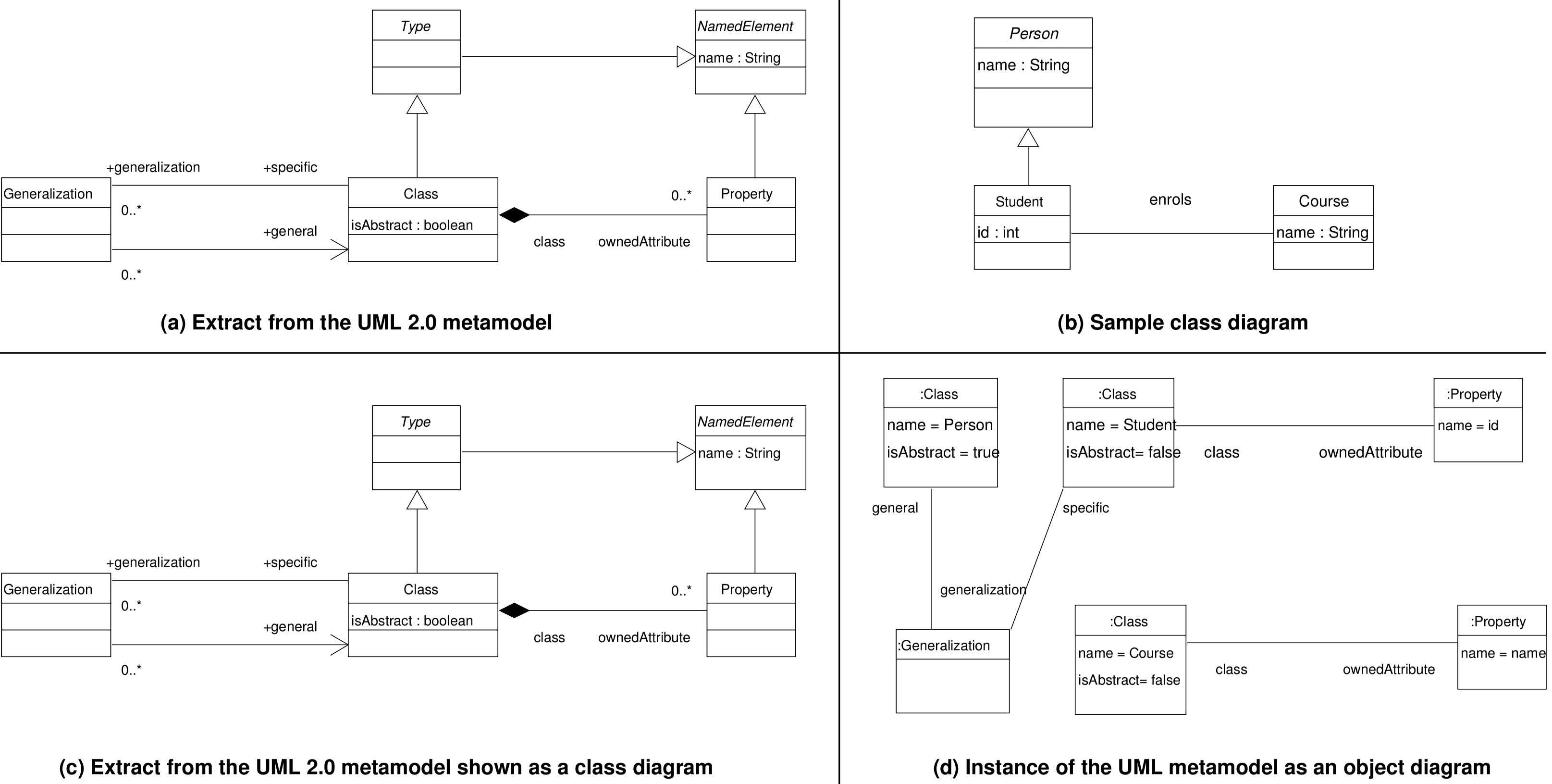

Figure 3.3: Representing metamodels as models. *This diagram shows how the UML metamodel and UML class diagram at the M2 and M1 layers of the OMG metamodelling hierarchy can be represented at the M1 and M0 layers. Although diagrams (a) and (c) on inspection look identical, please note that the elements in (a) are elements from the UML metamodel and reside at the M2 level and the elements in (c) are elements from a class diagram and reside at the M1 level.*

Thus we required an OCL tool with the ability to check and evaluate OCL expressions over UML models.

An example of how this works for the UML metamodel and a class diagram is shown in Figure 3.3. Figure 3.3 (a) represents the *M2* layer of the OMG metamodelling hierarchy and shows a simplified extract from the UML metamodel. A sample class diagram is shown in Figure 3.3 (b), it is an instance of the UML metamodel and represents the *M1* layer of the hierarchy. To bring this down one level in the metamodelling hierarchy, the UML metamodel is depicted as a class diagram as shown in Figure 3.3 (c) and is now at the *M1* level. The sample class diagram is now at the *M0* level and is shown here in Figure 3.3 (d) as a set of instances or objects of the elements in the class diagram of Figure 3.3 (c). The OCL queries corresponding to the metric definitions are then evaluated over these objects to compute the metrics.

As well as having support for OCL expressions over UML models we also require an OCL tool that has support for syntax checking and highlighting of OCL expressions to help ensure that the metrics specified are syntactically correct. As we are adhering to OMG standards, we also require full support of the newest version of the OCL 2.0 syntax. After considering the limited number of tools available Octopus was chosen. At the time of implementing the dMML tool, Octopus was one of the few tools available that supported the use of the OCL 2.0 syntax and satisfied all our requirements [Obj].

Octopus, an acronym for OCL Tool for Precise UML Specifications is an Eclipse plug-in developed by Klasse Objecten [Obj]. It offers two main functionalities. The first is the ability to check and identify errors in OCL expressions. It checks the syntax of expressions, as well as the expression types, and the correct use of elements of the UML class diagram such as attributes and association roles. Second, it provides the functionality to convert UML models, including OCL expressions, into Java code. Octopus also generates an XML reader and writer for a given UML/OCL model. The reader will read the contents of an XML file and produce Java objects which correspond to instances of the UML model.

The dMML tool builds upon and uses Octopus to both check the syntax of the metric definitions presented as OCL expressions and to translate the OCL expressions to Java code in order to automate the generation of a measurement tool from the metric definitions. At the time of implementing dMML, the most up to date version of Octopus was version 2.2.0 which runs in Eclipse 3.1 using Java 1.5.

Since then, later versions of Eclipse have been released, the most recent being the Ganymede, version 3.5 [Eclb]. Unfortunately, Octopus has not been updated for these later versions of Eclipse and as such the dMML tool is only available for version 3.1 of Eclipse.

3.3.1 Overview of dMML

An overview of how the dMML tool is used to define and calculate software metrics is shown in Figure 3.4. In this figure, shapes in blue represent in-house applications created as part of the system, shapes in pink are used to represent large scale independent applications or tools and the remaining items indicate the data that is used as input or output to the system described in the figure. This colour coding system is also applicable to the other figures in this thesis. The figure is divided into two layers, the upper layer represents the metric definition process, which is done once for each metric set. The lower layer represents the metric calculation process, where the metrics are applied to a set of metamodel instances.

The language metamodel describes the domain over which the metrics are to be

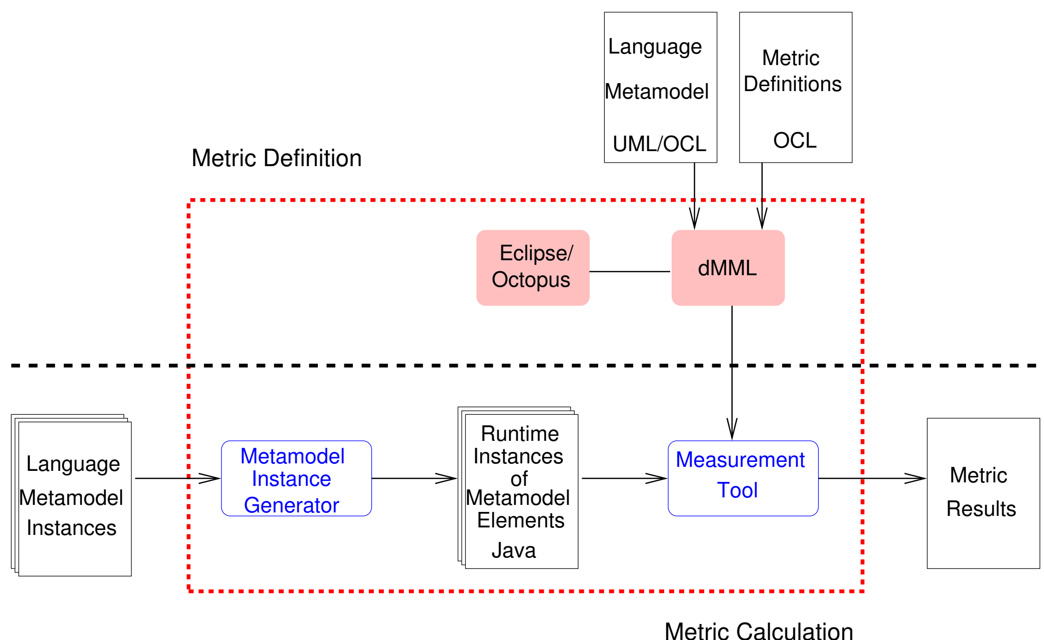

Figure 3.4: dMML - An environment for the definition of software metrics. *This system overview diagram shows the main inputs to and outputs from the dMML tool, which is implemented as an Eclipse plug-in.*

applied. The definition of this language metamodel is represented as a UML class diagram, and the corresponding Java representation of the metamodel is forward-engineered using Octopus.

The main tools that are developed are dMML for metric definitions, the *Measurement Tool* which is a Java program automatically generated by dMML for each metric set and the *Metamodel Instance Generator* which must be created by the user and is used for creating runtime instances of the language metamodel.

The red dashed line in Figure 3.4 delimits the system, and shows that its inputs are a language metamodel, a set of metric definitions in OCL and a set of metamodel instances. The output of the system is the set of metric values calculated by applying the metrics to the metamodel instances.

Using dMML to apply the metric definition and calculation approach to any metamodel or language is a four step process:

Step 1: Express the metamodel in UML and OCL. To perform this task the user loads the metamodel of the language or domain over which the metrics are to be applied. It must be possible to depict this metamodel as a class diagram for the approach to be applicable. This can be done either directly in the Octopus format or using a standard UML modelling tool that produces XMI which can then be imported into Octopus. The OCL constraints for the metamodel are then loaded into Octopus. The resulting model is then checked for correct use of OCL syntax and metamodel elements. Depicting MOF metamodels as class diagrams can be easily achieved using the UML profile for MOF [OMG04]. The OMG specification for MOF does not define a textual or graphical representation for MOF [OMG06a]. However, there is a UML Profile that defines a bi-directional mapping between the UML and MOF [OMG04]. The profile facilitates the creation of metamodels using the UML and the viewing of MOF metamodels. This can be used to express the metamodel in UML and OCL. For example, MOF classes map to UML classes, MOF attributes to UML attributes and vice versa. In addition, any constraints on the MOF metamodel map directly to UML constraints.

Step 2: Define the metrics as OCL queries over the metamodel. During this step the user creates and defines a set of metrics specific to the language or domain described by the metamodel loaded in step 1. dMML uses the Octopus plug-in to check these OCL expressions for syntax errors and incorrect use of model elements.

Step 3: Automatically generate a measurement tool from the metric definitions. Octopus is used to forward engineer a Java implementation of the UML/OCL specification including the metric definitions and the measurement tool is created as an independent Java application that can be invoked from the command line. This measurement tool makes use of the Java implementation generated by Octopus and a Builder class which is used to generate runtime instances of the language metamodel. A Java application must be created that generates instances of the language metamodel. This is achieved by creating a Metamodel Instance Generator class that extends the Builder class. To create an instance of the language metamodel, this class must make use of the Java classes (corresponding to the metamodel elements) produced in step 3. The name of this class is then specified in a dMML properties file for use by the measurement tool in step 4.

Step 4: Use the measurement tool to apply metrics to instances of the metamodel. The measurement tool can now be invoked from the command line (allowing metrics to be calculated in batch). Using reflection, the measurement tool reads the name of the Metamodel Instance Generator class from the dMML properties file and instantiates this class and uses it to create and load a runtime instance of the metamodel. If the instance of the metamodel is created successfully, the methods corresponding to the metric definitions are called, passing the relevant elements of the metamodel to the method and the metric results for that element are returned. These results are recorded and exported in text format.

To extend dMML to work with any language metamodel the user only needs to add the functionality to convert instances of the metamodel to Java runtime instances of the metamodel elements. dMML tool is parameterised by both the language metamodel and the definition of the metrics thus making it extensible to any language and any set of software metrics.

3.4 Using the Approach to Define and Calculate Software Metrics

In this section we demonstrate the feasibility of the approach by applying it to two different languages, the UML and Java.

3.4.1 An Illustration using the UML

In this subsection we describe how dMML was used to calculate the CK metrics for UML class diagrams using the UML 2.0 metamodel as a basis for the definitions. An outline of how dMML works for the UML can be seen in Figure 3.5. The language metamodel here describes the domain over which the metrics are to be applied, in this case UML 2.0 class diagrams. For reference, we have shown part of this metamodel in Figure 3.6 as taken from [OMG05b]. We developed a tool to create an instance of the part of the UML 2.0 metamodel relevant to class diagrams. We elaborate on each of the steps involved in applying the approach to UML class diagrams.

Step 1: Expressing the UML2.0 metamodel in UML and OCL

With reference to Figure 3.5, the Language Metamodel used is the standard metamodel for UML class diagrams [OMG05b]. For simplicity we chose to depict only the part of the UML metamodel relevant for defining the CK metrics. This part of the metamodel was manually written using the Octopus syntax for UML models. One of the obvious problems with using Octopus is taht it creates a Java implementation which does not directly support multiple inheritance. To overcome thos problem we use Java interfaces.

Step 2: Defining the CK metrics over the UML 2.0 metamodel

The CK metrics were expressed as OCL queries over the part of the UML 2.0 metamodel [OMG05b] that defines class diagrams. This is the first presentation of these definitions using the UML 2.0 metamodel. However, it was not possible to precisely measure all these metrics from a UML class diagram. Implementation details, such as the code in the bodies of method definitions are required to measure the CBO, RFC and LCOM metrics. However, it was possible to provide definitions to estimate the values for these metrics based on the information in the UML diagrams. Such measures are useful as they can provide upper and lower bounds for metrics calculated at later stages in the design or implementation process. In total, the definitions were composed of approximately 38 OCL queries and 93 lines of OCL.

As an example of the format of the CK metric definitions, Figure 3.7 illustrates how the NOC metric was expressed as an OCL query over the UML 2.0 metamodel. Here, the definition is parameterised by a single `Classifier` element, and the

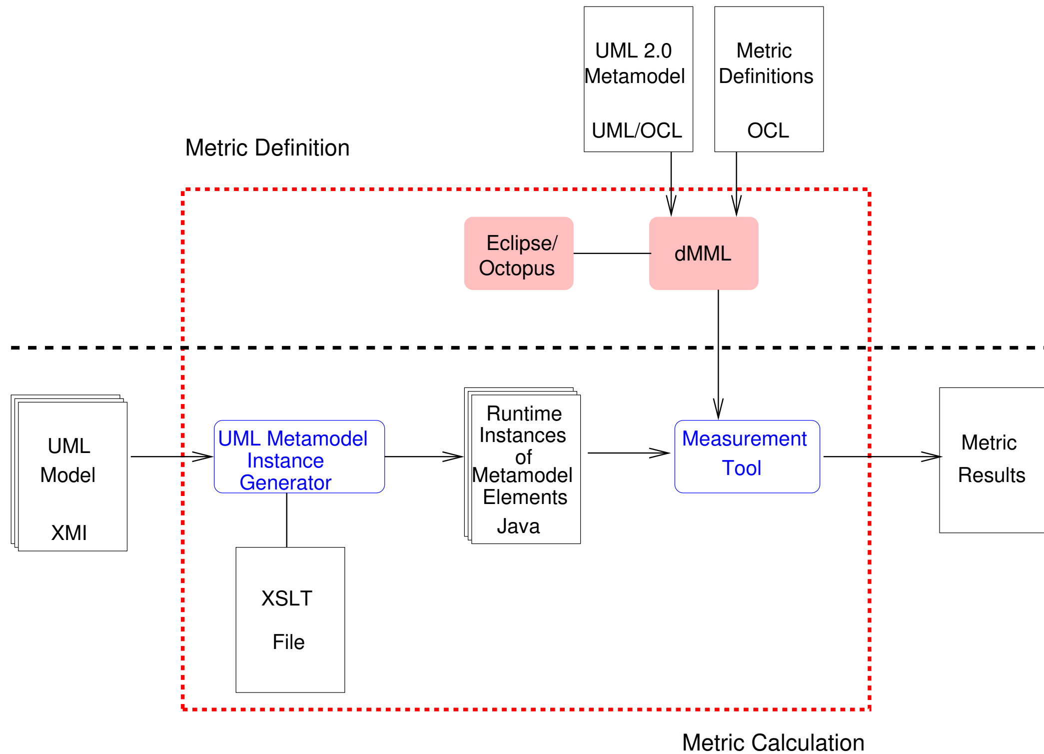

Figure 3.5: The use of dMML to define and calculate metrics for UML class diagrams. *This figure shows the two phases of our system: metric definition, centered on the dMML tool, and metric calculation, achieved by our UML metamodel instantiator program and an automatically generated measurement tool.*

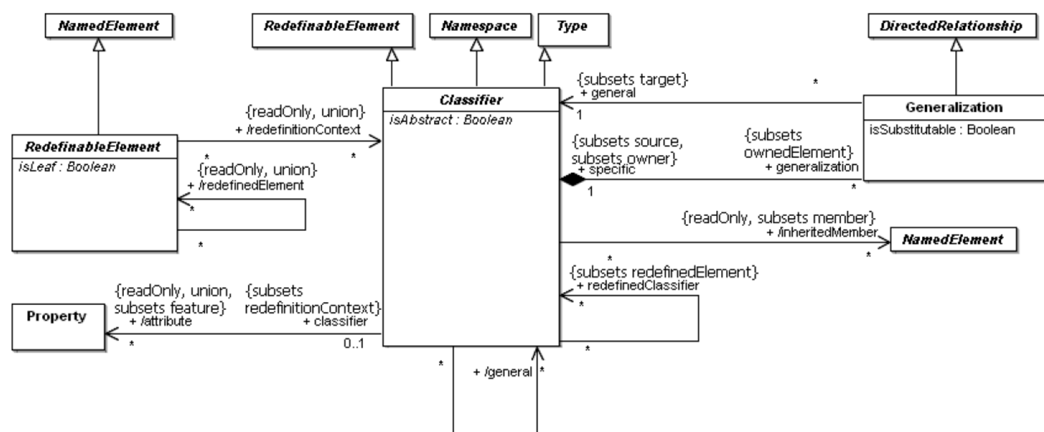

Figure 3.6: Excerpt from the UML2.0 metamodel. *This figure shows some of the main classes and relationships from the UML2.0 metamodel that are used in the OCL definitions [OMG05b].*

```
-- Returns the NOC value for the Classifier c
context CKMetrics::NOC( c : UML::Classifier ) : Real
body: self.children(c)->size()

-- Returns the set of all immediate descendents
-- of the Classifier c
def: children( c : UML::Classifier ) : Set( UML::Classifier )
=self.scope->excluding( c )->select( i:UML::Classifier | i.parents()
                                    ->includes( c ) )->asSet()
```

Figure 3.7: NOC Metric Definition using the UML metamodel. *This OCL code defines the* NOC *metric from the CK metrics set, and is part of a larger definition of the whole CK metric set which we have implemented using* dMML.

body of the definition returns the size of the set of all children of this class. The auxiliary operation `children` traverses the elements and relationships in the UML metamodel to assemble this set.

Step 3: Automatically generate a measurement tool from the CK metric definitions.

In total, the part of the UML 2.0 metamodel relevant for the CK definitions was represented using just under 2942 (non-blank, non-comment) lines of Java code. The CK metric definitions were implemented in 574 (non-blank, non-comment) lines of Java code. A metamodel instance in this case is an actual class diagram, represented in XMI, the standard output format for most UML modelling tools. Octopus was used to generate an XML reader for the UML model that represents the UML 2.0 metamodel. An XSLT transformation was written to convert class diagrams represented in XMI to XML files that can be understood by this XML reader.

Step 4: Applying the CK metrics to UML 2.0 metamodel instances

As a proof of concept, dMML was used to calculate the CK metrics for an open source project, *Velocity* which is part of the Apache Jakarta project. The Velocity project provides a Java-based template engine that can be used to reference objects defined in Java code [Jak]. Version 1.2 of *Velocity* was chosen as this is the version used in the study by Briand *et al.* [ABF04]. The project is distributed as a

singe JAR file which was reverse engineered using Rational Rose to obtain a UML model. The resulting model was exported in XMI format and used as input to the automatically generated measurement tool. The measurement tool using the UML metamodel instantiator developed in step 3, created runtime metamodel instances corresponding to the elements of the UML model and calculated and reported the CK metrics for each of these elements.

3.4.2 An Illustration using Java

In this subsection we describe how the dMML was used to calculate metrics for Java programs based on a Java metamodel. For consistency, the CK metrics were again chosen to illustrate this. An outline of how dMML works for Java can be seen in Figure 3.8. The domain over which the metrics are to be applied is Java, represented here by the Dagstuhl Middle Metamodel (DMM) [LTP04]. The DMM was designed as a language independent metamodel, as part of a project that aims to produce a de facto standard model for program entities.

Other parts of the system include dMML for metric definitions, and *Java to DMM* for converting Java class files to instances of the DMM, BCEL which is used by the *Java to DMM* tool and the *Measurement Tool*, which is a Java program automatically generated by dMML from the metric definitions.

Step 1: Expressing the DMM in UML and OCL

This was achieved by depicting the DMM as a UML class diagram using the Octopus syntax for UML models. There are no constraints in the DMM. Again, only the part of the DMM relevant for calculating the CK metrics were implemented in the Octopus syntax. Consequently, the classes in the `SourceObject` hierarchy that represent details about the code were not implemented as it appears in the original program. This does not preclude these classes being added later. The final representation of the DMM consisted of 19 classes from the `ModelObject` hierarchy.

Step 2: Defining the CK metrics over the DMM

We successfully produced definitions for the set of the CK metrics using the OCL and the DMM. This required approximately 25 OCL queries and 63 OCL lines in total. One advantage of our approach was that we could re-use much of the OCL from existing queries when defining new metrics. One example of this is when

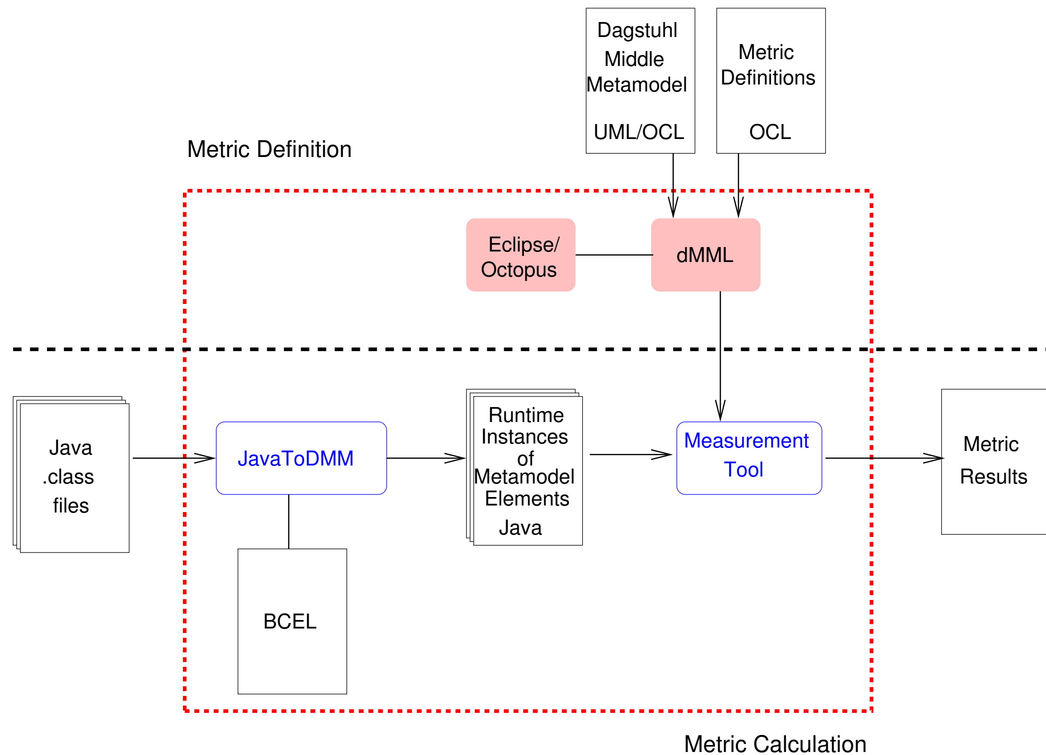

Figure 3.8: The use of dMML to define and calculate metrics for Java programs. *This figure shows the two phases of our system: metric definition, centered on the* dMML *tool, and metric calculation, achieved by our Java-to-DMM program and an automatically generated measurement tool.*

defining the *RFC* metric, it was possible to re-use the OCL from the *WMC* metric as both metrics need to calculate the number of implemented methods in a class.

To illustrate how the metrics were defined using the DMM, Figure 3.10 gives details of the RFC metric. The response set is the set of all implemented methods of this class and all methods invoked by this class. The relevant classes and associations from the DMM are shown in Figure 3.9 for reference. The RFC definition is parameterised by a single `Class`, and the body of the definition returns the size of the response set for this class. The auxiliary operation `methods-DirectlyInvoked(DMM::Class c)` gathers all methods invoked by each of the implemented methods in the class. The auxiliary operation `methods-DirectlyInvoked (DMM::Method m)` traverses the invokes association to gather all `BehaviouralElements` invoked by the method `m` and then selects all elements from this set that are methods.

To provide a comparison with the metrics presented for UML class diagrams,

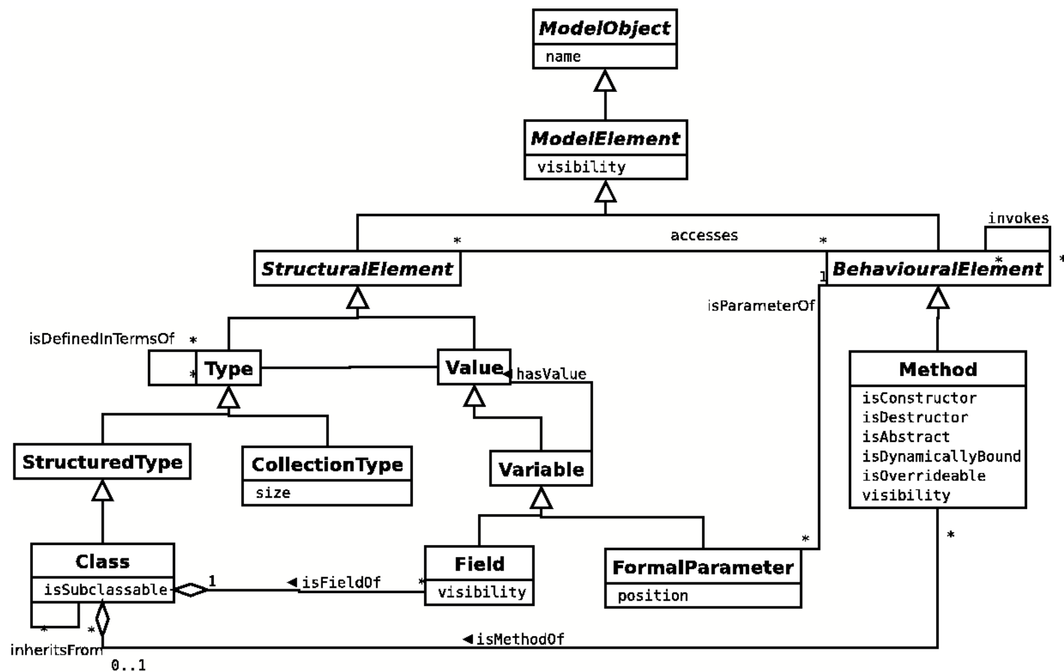

Figure 3.9: Excerpt from the Dagstuhl Middle Metamodel. *This figure shows some of the main classes and relationships from the DMM that are used in our OCL definitions [LTP04].*

we also give details of the NOC metric that was defined using the DMM in Figure 3.11.

Step 3: Automatically generate a measurement tool from the CK metric definitions.

In total, the DMM `ModelObject` hierarchy was represented using approximately 2266 (non-blank, non-comment) lines of Java code. The CK metric definitions were implemented in 452 (non-blank, non-comment) lines of Java code. A tool was implemented to convert Java programs to instances of the DMM metamodel. To achieve this, it was necessary to read in Java programs and to instantiate the DMM classes produced in Step 1. There are a number of different possible ways of doing this, including parsing the Java source code or processing a compiled `.class` file directly. We chose the second option as the contents of the `.class` file most closely resembled the information needed to instantiate the DMM implementation. In particular, access relationships between classes arising from the use of fields and variables in a method are easy to identify at the bytecode level, since they are

```
-- Returns the RFC value for the Class c
context CKMetrics::RFC( c : DMM::Class ) : Real
body: self.implementedMethods( c )
      ->union( self.methodsDirectlyInvoked( c ) )->asSet()->size()

-- Returns the set of methods implemented in the Class c
def: implementedMethods( c : DMM::Class ) : Set( DMM::Method )
= c.hasMethods-self.abstractMethods( c )

-- Returns the set of abstract methods in the Class c
def: abstractMethods( c : DMM::Class ) : Set( DMM::Method )
= c.hasMethods->select( m : DMM::Method | m.isAbstract )

-- Returns a set containing all methods directly invoked by all
-- the methods implemented in the Class c
def: methodsDirectlyInvoked( c : DMM::Class ) : Set( DMM::Method )
= self.implementedMethods( c )
  ->collect( m : DMM::Method | self.methodsDirectlyInvoked( m ) )
  ->flatten()->asSet()

-- Returns a set containing all methods directly invoked
-- by the method m
def: methodsDirectlyInvoked( m : DMM::Method ) : Set( DMM::Method )
= m.invokes->select( be : DMM::BehaviouralElement
                   | be.oclIsTypeOf( DMM::Method ) )
                   ->collect( belem : DMM::BehaviouralElement |
                              belem. oclAsType( DMM::Method ) )
                              ->asSet()
```

Figure 3.10: RFC Metric defined using the DMM. *This OCL specification defines an operation to calculate the RFC metric for a class, as well some auxiliary operations. The entities used in the definition are from the DMM [LTP04].*

translated into a single bytecode instruction.

Our implementation uses the Apache Bytecode Engineering Library (BCEL) to read in and traverse the contents of the `.class` file. The BCEL API provides classes representing the contents of the `.class` file, and methods to access classes, fields, methods and bytecode instructions [BCE]. Using the BCEL it was relatively easy to traverse these structures and instantiate the DMM, and required less than 600 (non-blank, non-comment) lines of Java code. It should be noted that using BCEL would not be suitable for a more detailed representation than the DMM `ModelObject` hierarchy. Source level details such as Java statements (e.g. while and for loops) are not represented in the bytecode, and tables giving local variable

```
-- Returns the NOC value for the Class c
context CKMetrics::NOC( c : DMM::Class ) : Real
body: self.children(c)->size()

-- Returns the set of all immediate descendents of the Class c
def: children( c : DMM::Class ) : Set( DMM::Class )
= c.subclass
```

Figure 3.11: NOC Metric Definition using the DMM. *This OCL specification defines an operation to calculate the NOC metric for a class. This definition is provided here for comparison with the NOC metric for UML class diagrams shown in Figure 3.7.*

names and mappings to lines of Java code are optional at the `.class` file level.

Step 4: Applying the CK metrics to DMM instances

We applied the measurement tool generated in step 2 to programs from the open source project Velocity, version 1.2. We included constructors, mutators and accessor methods as ordinary methods, but excluded attributes and methods that are inherited but not defined in a class. The CK metrics were calculated on all the class files contained in the velocity JAR file.

3.5 Discussion

While the approach outlined in this chapter provides a way to define software metrics using a standard language and facilitates the automatic generation of measurement tools directly from the software metric definitions, it does not completely fulfill the main aims set out in Chapter 1.

Numerous software metrics exist in the literature and many of these metrics are applicable to a number of different models of a software system. The disadvantage of using metamodel-specific metrics as outlined in this chapter is that it is difficult to re-use metric definitions. For example, in this chapter two definitions were required for each of the metrics from the CK metric set, one for the UML and one for Java. Similarly, if we want to define these metrics for other object-oriented languages, for example C++ or C# then we need to define the metrics again and again. Ideally it should be possible to define a set of metrics once, and then adapt them to each relevant metamodel in turn. This provides not only for economy of expression but

also helps to ensure that the same concepts are being measured from the different models. It is our view that this is best addressed by specify the metrics in a generic way, independent of the particular model [MP07]. Like Mens and Lanza, we believe that this can be achieved using a *language-independent*, *metrics-specific* metamodel [ML02].

Furthermore, using the approach outlined in this chapter required the implementation of two separate programs for instantiating the metamodels over which the metrics were being applied, thus adding an extra level of complexity to the measurement approach. One way to address the complexity of software development is using the principles of abstraction and problem decomposition which can be realised using modelling and model transformations [SK03]. Our proposal is to use a model transformation to describe the mapping from the language under measurement to a *language-independent*, *metrics-specific metamodel*. In this case, the use of a model transformation would raise the level of abstraction by simply specifying what in the language metamodel maps to what in the measurement metamodel without having to be concerned with the implementation details.

3.6 Summary

In this chapter, the need for a clear and precise approach to defining software metrics has been addressed by exploiting the OCL as a specification language, and harnessing language metamodels to provide an approach to defining metrics. While the approach to date is similar to other research in this area, particularly that of Baroni *et al.*, it differs in a number of key areas. First, the approach has been generalised at the metamodel level and applied to various modelling languages. Second, the metric definition and calculation procedure is highly extensible, allowing for different versions of the same metric to be easily implemented and compared.

A tool, dMML, has been implemented that uses the Octopus tool to translate OCL metric definitions into Java code that calculates the metrics for metamodel instances. To demonstrate this approach, definitions of the CK metrics have been defined over both the UML metamodel and a Java metamodel, the DMM, and the resulting measurement tool has been run over a set of real-world programs thus demonstrating the robustness of this automatically generated measurement tool. A further contribution of this work is that it provides a first ever definition of the Chidamber and Kemerer metrics suite using both the UML 2.0 metamodel and a

Java metamodel as a basis for the definitions.

In summary, applying the approach to both UML class diagrams and Java programs, has demonstrated both the feasibility and generalisability of this approach to defining software metrics. However, there are a number of issues such as language dependent metric definitions, complexity of creating an implementation to instantiate the language metamodel and the issue of the validity of the measurement results produced by the approach. The remainder of this thesis is concerned with addressing these issues.

Chapter 4

A Metamodel for the Measurement of Object-Oriented Software

In this chapter the limitations of the approach outlined in the previous chapter are addressed by developing a MOF-compliant, language-independent metamodel for defining object-oriented software metrics and demonstrating how the dMML tool uses it to automatically generate a measurement tool that calculates a set of coupling and cohesion metrics and the CK metrics set. Details of the work described in this chapter have been presented in McQuillan and Power [MP08b, MP].

4.1 Introduction

To support the definition and implementation of language-independent object-oriented software metrics we develop a metamodel for defining and calculating coupling and cohesion metrics, which we call the *measurement metamodel*. This metamodel is based on a standard terminology and formalism outlined by Briand *et al.* when defining their coupling and cohesion measurement frameworks [BDW98, BDW99]. However, developing and working with metamodels can be difficult since they deal with abstract concepts and therefore it is important to ensure that this measurement metamodel is correct. By correct, we mean that the metamodel specification is consistent, is neither under or over-constrained and adequately describes what the user intends. It is also important that the metric definitions themselves and the automatically-generated measurement tool are also correct. Any errors or ommis-

sions would have a fundamental impact on the correctness and reliability of the metrics calculated using this approach.

In a wider context, it is important to be able to ensure the correctness and quality of any software application that is based around a metamodel, for example UML modelling, model transformation and code generation tools. One way to achieve this is through software testing. When metrics are defined directly over UML or Java metamodels they can be tested using the wide range of readily-available UML models and Java programs. However, using the measurement metamodel makes testing difficult, since there is not a ready pool of example metamodel instances. Furthermore, there is no direct way of automatically generating metamodel instances for use as test inputs when testing metamodel-based software applications [EKTW06].

In this chapter an approach to the analysis of MOF-compliant metamodels is described and applied to the measurement metamodel. In this approach, we express the metamodel using the UML and the OCL. We generate a specification in the Alloy language corresponding to the metamodel [Jac06], and use this to examine the metamodel constraints and to generate sample instances of the metamodel. We have created a *reflective instantiator* that takes these Alloy generated models and transforms them into instances of a Java implementation of the measurement metamodel, thus harnessing Alloy's lightweight approach to generate a test suite for the generated measurement tool. We use this test suite to determine if the tool correctly computes metric values for the coupling, cohesion and the CK metrics sets. Finally, we evaluate the adequacy of the generated test suite with respect to code coverage.

4.2 The Measurement Metamodel

In this section we present a metamodel for coupling and cohesion measurement and illustrate how it is used for the definition and calculation of object-oriented metrics.

4.2.1 Structure of the Metamodel

One requirement of our metamodel is that it is interoperable with the UML and Java metamodels and thus it has been developed to conform to the MOF. This ensures that all three metamodels are specified using the same formalism, thus facilitating the translation of instances of the UML and Java metamodel to instances of the

MOF Element	Number
Class	9
Constraint	15
Enumeration	2
Generalisation	7
Association	16
Property	5
Operation	27
Parameter	4

Table 4.1: Summary of the size of the measurement metamodel. *This table gives a summary of the size of the metamodel in terms of the different types of MOF element that constitute the metamodel.*

measurement metamodel presented in this chapter. Moreover, our MOF-compliant metamodel is based on the standard terminology and formalism for coupling and cohesion measurement proposed by Briand *et al.* [BDW98, BDW99]. To create the MOF metamodel we examined the formalism and metric definitions provided in [BDW98, BDW99] and extracted a set of elements and relationships required for describing coupling and cohesion metrics and formulated these concepts as a MOF metamodel (see Section 4.3). The resulting metamodel captures the basic structure of an object-oriented system and specifically the concepts and relationships required for coupling and cohesion measurement. Table 4.1 summarises the basic details of the measurement metamodel in terms of the number of MOF elements used to construct the measurement metamodel.

The metamodel is composed of a single package called `MM` (Measurement Metamodel) and the contents of this package are shown in Figure 4.1 as a class diagram. This figure shows the main classes involved in the metamodel, along with the important associations necessary for distinguishing the different types of coupling and cohesion metrics. The central classes for coupling and cohesion metrics are `Class`, `Method` and `Attribute`. A `Class` is generalised by `Type`, which also generalises built-in types (e.g. `integer`, `string`) and user-defined types (e.g. `struct`, `enumeration`).

In order to implement the metric definitions, we distinguish between declared and implemented attributes and methods based on whether they physically appear in the class definition, or whether they are just present due to inheritance. Similarly, we partition methods into three types: those that are inherited without change, those

that are inherited and overridden, and those that are declared for the first time in a class. Other classifications of methods are given by the attributes of the `Method` class, for example a method is either public or not public and is either abstract or not abstract. A method is also tagged with an indicator of the purpose of the method. A method can be either a constructor, destructor, accessor or mutator. If the method does not fall into one of these four categories then it is tagged as general.

Another important concept necessary for coupling metrics is the notion of a method call or invocation which is represented by the `Invocation` class in the measurement metamodel. An invocation can be either static or polymorphic and the `Invocation` class has an attribute `type` to indicate this.

Finally, the elements `Type`, `Method`, `Attribute` and `FormalParameter` have a name associated with them to facilitate their identification when metrics are computed for them. This is represented using an abstract class `NamedElement` with a property `name` and a generalisation relationship between this abstract class and the elements `Type`, `Method`, `Attribute` and `FormalParameter`. For the sake of readability, we have omitted the `NamedElement` class from figure 4.1.

The metamodel specification also contains constraints, which specify semantic and syntactic properties of the data described by the metamodel. These constraints are specified in OCL and referred to as well-formedness rules. In total, the metamodel has 15 well-formedness rules. Further details of these are given in Section 4.4 and a complete specification of the metamodel can be found in Appendix A.

As our metamodel has been designed to be language-independent we believe that it can be easily applied to different object-oriented languages. Applying the metamodel to an object-oriented language involves mapping the different concepts of that language to the equivalent concepts in the measurement metamodel. In certain instances concepts of the language may not map directly to elements in our metamodel and in this situation a decision will have to be made as to which elements in the metamodel these concepts should map to. This mapping between the language and the measurement metamodel can be clearly defined using a transformation language and provides an automated way to transform models written in the language to instances of the measurement metamodel thus facilitating the automatic calculation of the metrics for that language. We discuss this further in Chapter 5. The metamodel may still need some work to apply to different styles of object-oriented language. But we do believe that it is generic in terms of representing the concepts for coupling and cohesion in an object-oriented system.

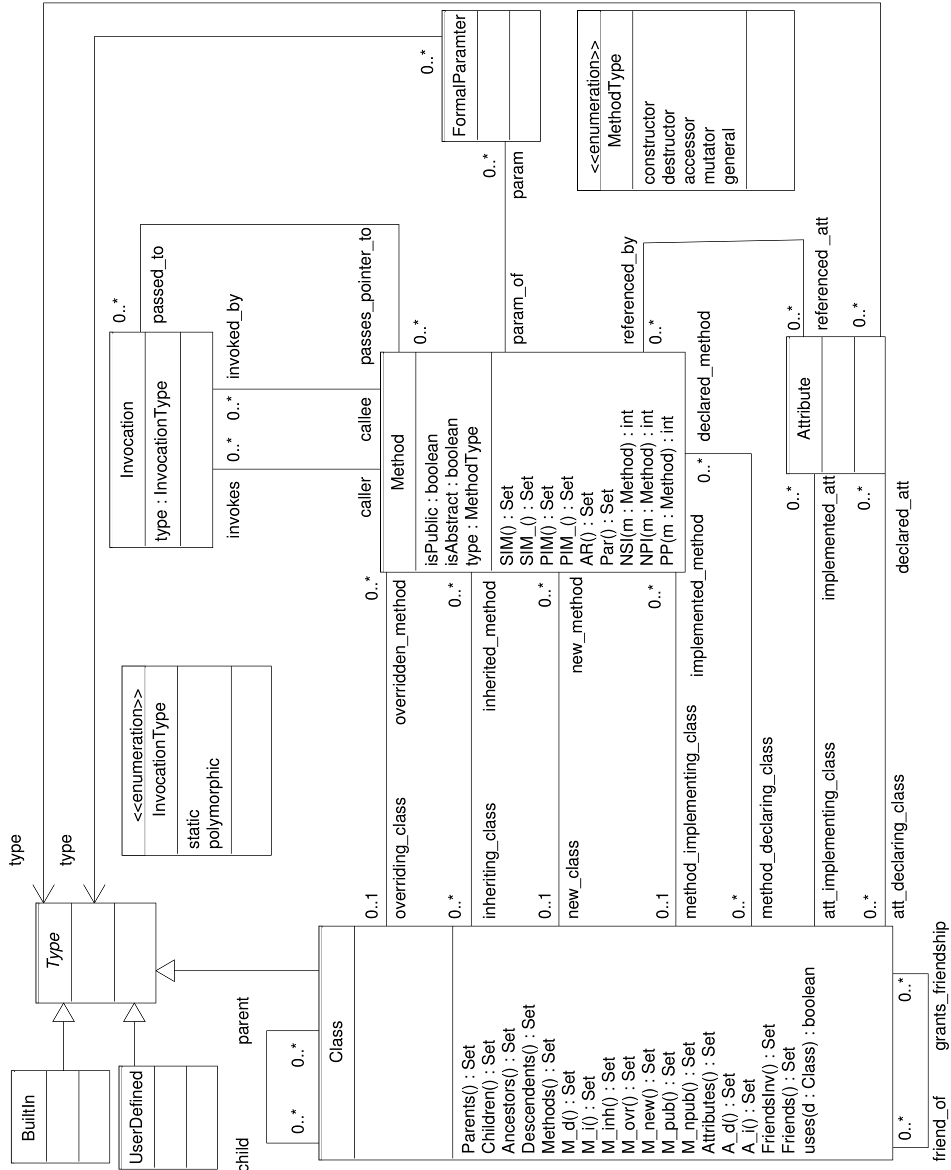

Figure 4.1: The measurement metamodel. *This figure shows the main classes and relationships in the measurement metamodel depicted as a UML class diagram. For clarity the NamedElement class has been omitted from this diagram.*

4.2.2 Defining Metrics using the Metamodel

In this subsection we describe how we used the metamodel to define three sets of existing object-oriented software metrics. The three sets of metrics *CKMetrics*, *Cohesion* and *Coupling* were taken from [CK94, BDW98, BDW99], respectively. In total, we defined 44 metrics and these are summarised in Table 4.2. These metrics have been discussed in more detail in Chapter 2.

In keeping with the approach outlined in Chapter 3, the metrics were defined as OCL queries over the measurement metamodel. The metamodel was extended with a separate metrics package containing a single class called `Metrics`, and each set of metrics was defined as follows:

1. A class was created in the metrics package for the metric set; this class extends the `Metrics` class. Any common auxiliary operations were defined in this class.
2. For each metric, an operation was declared in the class, parameterised by the appropriate metamodel elements.
3. The metrics were defined by expressing them as OCL queries using the OCL `body` expression.

As an example of a definition, Figure 4.2 presents the definition of the **coupling between objects** (CBO) metric. Here, the definition is parameterised by a single `Class`, and the body of the definition returns the size of the set of all classes in the system that have a *uses* relationship with that class. The auxiliary operation `uses` is defined as a query operation of the `Class` element of the measurement metamodel and is parameterised by a single `Class d`. This operation returns true if any of the implemented methods of the `Class` element polymorphically invokes

Metric Set	Metrics from references [CK94, BDW98, BDW99]
CKMetrics	WMC, NOC, DIT
Cohesion	LCOM1, LCOM2, LCOM3, LCOM4, LCOM5, NewLCOM2, NewCoh, Co, NewCo, TCC, LCC, ICH
Coupling	RFC, RFC', CBO, CBO', MPC, COF, DAC, DAC', ICP, IH_ICP, NIH_ICP, IFCAIC, ACAIC, OCAIC, ACMIC, OCMIC, IFCMIC, AMMIC, OMMIC, IFMMIC, FMMEC, DMMEC, OMMEC, FCMEC, DCMEC, OCMEC, OCAEC, FCAEC, DCAEC

Table 4.2: Summary of implemented metrics. *This table lists all 44 metrics that were defined and implemented using the measurement metamodel.*

```
-- Returns the set of classes that have a uses relationship
-- with the class c
context Coupling::CBO(c:MM::Class) : Real
body:MM::Class.allInstances()->excluding(c)
                             ->select(d:MM::Class |
                                      c.uses(d) or d.uses(c))
                                      ->size()

-- Returns true if the class uses class d
context Class::uses(d:Class) : Boolean
body: not self.implemented_method->collect(m:Method|m.PIM())
                            ->intersection(d.implemented_method)
                            ->isEmpty()
     or
     not self.implemented_method->collect(m:Method|m.referenced_att)
                            ->intersection(d.implemented_att)
                            ->isEmpty()

-- Returns the set of methods polymorphically invoked
-- by the method
context Method::PIM() : Set(Method)
body:self.poly_invoked()->asSet()

-- Returns a bag of methods polymorphically invoked
-- by the method
def:poly_invoked():Bag(Method)
=self.invokes->select(i:Invocation |
                      i.type = InvocationType::polymorphic)
                      ->collect(j:Invocation | j.callee)
```

Figure 4.2: Definition of the *CBO* metric in OCL using the measurement metamodel. *This OCL code defines the CBO metrics from the Coupling metric set, and is part of a larger definition of a set of Coupling metrics which has been defined and implemented using the measurement metamodel.*

an implemented method or references an implemented attribute of the `Class d`. The operation gathers the relevant attributes and methods of these two Classes by traversing the appropriate associations of the metamodel and calling the predefined `PIM` operation.

The operation `PIM` is defined as a query operation of the `Method` element of the measurement metamodel and returns all unique methods polymorphically invoked by that `Method`. It calls the `poly_invoked` operation on the `Method` to get a list of all polymorphically invoked methods of the `Method`. A method can make multiple calls to the same method so `poly_invoked` returns a bag of `Methods`

thus allowing the same method to appear more than once in the returned list. `PIM` returns this list of methods as a set thus removing any duplicate methods. The operation `poly_invoked` returns all polymorphically invoked methods of a `Method` by traversing the *invokes* relationship to gather all the `Invocation`s in which that method is the calling method. It then selects and traverses those `Invocation`s that are polymorphic and collects the methods that are being called or invoked by the `Method`.

Briand *et al.* suggest that other interpretations of this metric are also possible, for example the *uses* relationship may consider only statically invoked methods instead of polymorphically invoked [BDW99]. Defining and implementing this alternative definition of the CBO metric is rather straightforward using our measurement approach. The only change necessary is to replace the call to `PIM` with a call to `SIM` in the definition of `uses`. The query `SIM` returns the set of statically invoked methods of a given method in a similar way to how `PIM` collects the polymorphically invoked methods. No change is necessary to the implementation of the metric as this gets generated automatically from the definition.

After defining all 44 metrics shown in Table 4.2, dMML was used to automatically create a measurement tool to calculate these metrics. In Section 4.5, we report on how we evaluate the correctness of this measurement tool.

4.3 A General Approach to Analysing MOF Metamodels

Although the first step is concerned with defining a measurement metamodel for sofware measurement, it is also imperative to analyse the meatmodel to ensure it is correct. In this section an approach that can be used for the analysis of MOF-compliant metamodels is presented. As Alloy is core to this approach, we begin by briefly reviewing some of the important concepts of the Alloy language and analyser and then present an overview of the approach. The application of the approach to the measurement metamodel is described later in Section 4.4.

4.3.1 The Alloy Language and Analyser

The Alloy language and analyser has been used primarily to explore abstract software models and to assist in finding and correcting flaws in these models [Jac06].

Moreover, it has been successfully used for modelling and analysis in several different real-world domains including safety-critical systems [Den03] and the semantic web [DSW03]. We extend this domain of application by showing that Alloy can also be successfully applied to the the domain of MDE by using it to develop and analyse MOF-compliant metamodels.

Alloy is a formal specification language that has been developed by Daniel Jackson and his colleagues at MIT [Jac06]. It is based on typed first-order relational logic. An Alloy specification is based around *signatures* and *formulas* such as *facts*, *predicates* and *assertions* which are described next.

- **Signatures:** Signatures are used for defining the entities of a model and consist of a set of declarations that define the relations and operations of the entity.
- **Facts:** Facts impose constraints on a model. They are always enforced and must always hold true for all instances of a model.
- **Predicates:** Predicates are used to impose constraints on particular instances of a model. They may or may not be forced to hold true for a model.
- **Assertions:** Assertions follow from the facts of a model and specify the constraints of the model that are assumed or claimed to be true. They are never enforced for a model.

A fully automatic tool, called the *Alloy Analyser* has been developed simultaneously with the Alloy language. This is a "model-finder" tool that uses a constraint solver to analyse models written in Alloy. There are two types of analysis offered by the tool, namely *simulation* and *checking*. *Simulation* involves confirming the consistency of a predicate by generating a model instance that satisfies the predicate. *Checking* involves assessing the validity of assertions by finding counterexamples to the assertion. To make instance finding feasible, a user must specify a *scope* for the model under analysis. The *scope* puts a bound on how many instances of an entity may be observed in a model instance and thus limits the number of model instances to be examined.

The main concepts of Alloy are briefly illustrated with an example adapted from an Alloy tutorial [WTH]. The example is depicted in Figure 4.3 and shows a partial

```
sig Queue { root: lone Node }
sig Node { next: lone Node }

fact nextNotReflexive { no n:Node | n = n.next }

assert allNodesBelongToSomeQueue {
       all n:Node | some q:Queue | n in q.root.*next
}

pred show() {}

run show for 2

check allNodesBelongToSomeQueue for 2
```

Figure 4.3: Alloy specification for the Queue model. *This is an example of an Alloy specification and defines the Queue data structure.*

Alloy model specification for the Queue data structure. The Alloy code declares two signatures representing the entities `Queue` and `Node` in the model, along with their relations. In the declaration of `Queue`, the `lone` keyword specifies that the `root` relation maps each object of `Queue` to either zero or one object of `Node`. The model specification also contains a single fact that states that there is no `Node` that is equal to its successor.

It is possible to use Alloy to search for specific model instances that satisfy certain properties or constraints. This is done by formulating the constraint as a predicate and using Alloy to search for a model instance that satisfies this predicate. Alloy indicates and displays if an instance is found. However, if no instance is found no conclusion can be drawn as an instance may exist outside of the specified scope. In this example we want to demonstrate the consistency of the model, so therefore we want any instance of the model. To specify this we have written a predicate, `show()` that is empty and contains no additional constraints. To obtain a sample instance of the predicate the `run` command is used with a specific scope. Here, the scope is 2 which means Alloy will search for instances with at most two instances of each top level signature (i.e. at most two Queues and at most two Nodes). If an instance is found, Alloy indicates this and can be used to display the instance. An example of a model instance produced by Alloy that satisfies the predicate `show` is shown in (a) of Figure 4.4.

Additionally, to check an assertion with Alloy, the command `check` is used and

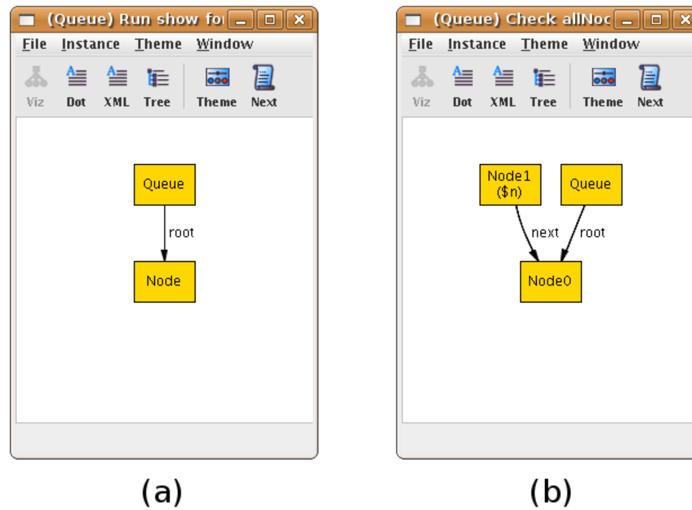

Figure 4.4: Output from the Alloy Analyser produced during the analysis of the Queue specification *On the left is a model instance that satisfies the predicate* `show` *and an the right is a counterexample to the assertion* `allNodesBelongToSomeQueue`.

---

as for the `run` command a scope must be specified. An example of an assertion, `allNodesBelongToSomeQueue` is also shown in Figure 4.3. This assumes that all Nodes belong to a least one Queue. To check this assertion Alloy searches all valid model instances with at most two instances of each signature for a counterexample i.e. a model instance that violates the assertion. If it finds one then the assertion is not valid and the counterexample is displayed. Otherwise the assertion may or may not be valid as a counterexample may exist outside of the specified scope. In the example shown here, Alloy has found a counterexample to the assertion and this is shown in (b) of Figure 4.4.

Recently, the parallel between specification in Alloy and modelling in UML has been noted by Massoni *et al.* [MGB04] and exploited by Anastasakis *et al.* [ABGR07]. Anastasakis *et al.* present a tool called *UML2Alloy*, that takes a UML class diagram, along with the associated OCL constraints, and translates this UML model into a corresponding Alloy model specification. The sample instances generated by the Alloy Analyser then correspond to object diagrams from the UML model. However, their tool does not provide any automated handling of the generated Alloy instances.

Several other researchers have used Alloy to analyse and reason about metamodels. For instance, an alternative definition of the UML metamodel is presented in [NW02] and analysed using Alloy. Zito *et al.* use Alloy to formalise and analyse the

package merge concept of the UML 2.0 metamodel [ZD06]. Both these approaches are similar to the one presented in this chapter in that they use Alloy to describe a *meta*model, as opposed to a *model* as with Anastasakis *et al.*. However, the main focus of their research has been on the analysis of the UML metamodel. The work described in this chapter is concerned with using Alloy to analyse a metamodel for object-oriented software measurement.

4.3.2 Overview of the Analysis Approach

An overview of our approach is depicted in Figure 4.5. In this figure, the system is delineated by a dashed red line. The inputs to the system are the metamodel and its constraints expressed as UML and OCL, and are shown on the left of the figure.

Both Octopus [Obj] and Alloy are third-party tools used in our system. The metamodels used in our measurement approach are represented as UML class diagrams in a format specific to the Octopus tool. Therefore, the *UML2Alloy* tool used here is a re-implementation of the same tool of Anastasakis *et al.* [ABGR07], but specialised for Octopus. The process is almost fully automated, with user intervention limited to providing the original UML/OCL description of the metamodel, and examining the generated Alloy specification along with the instances of the metamodel produced by Alloy. This is depicted by the green stick-figure in Figure 4.5.

There are three main steps in this process:

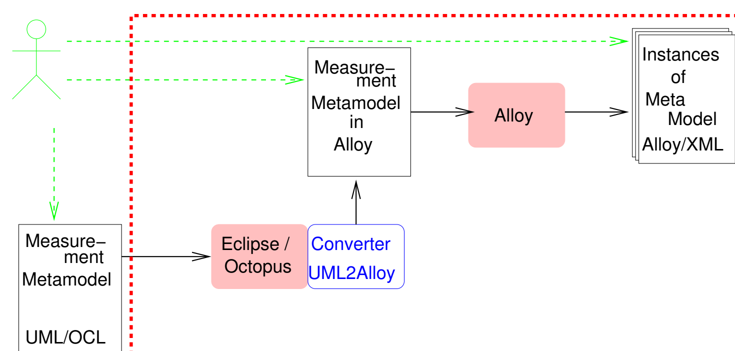

Figure 4.5: Overview of the approach to analysing the MOF-compliant metamodel. *The elements in the system are enclosed by a dashed red line. The input, shown on the left, is the metamodel and its constraints expressed using UML and OCL.*

Step 1: Expressing the metamodel in UML and OCL. This process is not automated, the metamodel is depicted as a class diagram using the UML profile for MOF [OMG04], either directly in the Octopus format or using a standard UML modelling tool that produces XMI which can then be imported into Octopus. The OCL expressions for the metamodel are then loaded into Octopus. The resulting model is then checked for correct use of the OCL syntax and metamodel elements using the Octopus tool.

Step 2: Transforming the metamodel to Alloy. We have implemented a tool to convert an Octopus UML/OCL metamodel to an Alloy model specification. Since this tool mimics the operation of the *UML2Alloy* tool of Anastasakis *et al.* [ABGR07], we have not given it a distinctive name. In what follows we briefly outline the transformation approach. Each class in the metamodel is mapped to an Alloy signature. All attributes of a class are mapped to fields of the corresponding Alloy signature. Enumerators are mapped to abstract signatures in Alloy, with enumerator literals mapped to sub-signatures that extend the abstract signature of the enumeration. The associations in the metamodel are also mapped to attributes in the appropriate Alloy signatures. Association multiplicities, 1, 0..1, 0..`*`, and 1..`*` map to the keywords `one`, `lone`, `set` and `some`. An additional fact is generated in the Alloy specification for bi-directional UML associations to show that the relations are symmetric. The basic data types are mapped to equivalent signatures from the Alloy library. Finally, any constraints on the metamodel in the form of OCL invariants are mapped directly to Alloy facts. At present, the OCL map does not cover the full language, and requires some user intervention for more difficult constructs.

Step 3: Analysis of the metamodel. The Alloy Analyser is used to analyse the Alloy specification to detect flaws in the metamodel specification. For example, it can be used to generate Alloy model instances (or metamodel instances) that conform to the well-formedness rules of the metamodel. If an Alloy model instance cannot be found then there is an inconsistency in the metamodel specification. It is also possible to check if the metamodel is over-constrained by specifying certain required properties and generating Alloy model instances that satisfy these properties. Similarly, the metamodel can be analysed to assess if it is under-constrained by enumerating and exploring all possible Alloy model instances that can be generated from the Alloy specification of the metamodel. This is useful to identify invalid

metamodel instances i.e. instances that do not represent what the user intends their metamodel specification to represent.

4.4 Development and Analysis of the Measurement Metamodel

While the approach outlined in Section 4.3 will work for any MOF-compliant metamodel, our original intention was the specification and analysis of a metamodel for coupling and cohesion measurement. In this section we elucidate our approach using that measurement metamodel.

4.4.1 Applying the Approach to the Metamodel

As described in Section 4.3, the first step of our approach is to express the metamodel in UML and OCL. As we were basing our metamodel on that of Briand *et al.*, we began by expressing the concepts described by Briand *et al.* [BDW98, BDW99] as a class diagram and formalised any well-formedness rules that were expressed in natural language by Briand *et al.*. An example of such a rule is that *the set of all new, overriding and inherited methods of a class are disjoint*. We suspected that all these constraints were not sufficient to describe our metamodel and thus added 15 more constraints to the original set of 12 constraints to give a total of 27 well-formedness rules. Once we had formalised all of the rules in OCL, we used Octopus to statically check the OCL constraints and then translated the MOF-compliant metamodel and its well-formedness rules to Alloy.

An example of the translation of UML classes to Alloy is shown in Figure 4.6. This figure gives the Alloy specification for the `Class` element of the measurement metamodel which is defined in Alloy as a signature extending the `Type` signature. The associations for a class are represented by fields, which are shown here in four groups. These groups represent inheritance relationships, friendship relationships (for C++), and the relationships with the class' attributes and methods. Each attribute of a class is either declared or implemented in that class. Each method is either declared or implemented in the class, and is either a new, overridden or inherited method.

Furthermore, any constraints on the metamodel in the form of OCL invariants were mapped directly to Alloy facts. An example of such a constraint is depicted

```
sig Class extends Type
{
  /* Inheritance */
  parent: set Class,
  child: set Class,

  /* Friendship */
  grants_friendship: set Class,
  friend_of: set Class,

  /* Class - Attribute Relationships */
  declared_att: set Attribute,
  implemented_att: set Attribute,

  /* Class - Method Relationships */
  declared_method: set Method,
  implemented_method: set Method,
  new_method: set Method,
  overridden_method: set Method,
  inherited_method: set Method
}
```

Figure 4.6: Alloy signature for the element `Class` of the measurement metamodel. *This is a representation of the element `Class` in the Alloy specification language.*

in Figure 4.7. This invariant states that *if a class does not have any parents then it cannot have any overridden methods* and maps to a fact in the Alloy specification.

To reduce the number of Alloy generated instances to be examined during the analysis and test case generation stages we omitted the `NamedElement` concept from the Alloy specification of the metamodel. We believe this does not effect the generalisablity of our results as this element is only used for identification purposes during software measurement and is not involved in any of the constraints of the metamodel or metric definitions. The Alloy specification of the measurement metamodel was then analysed to check if it was consistent and was neither under- or over-constrained. The results of this analysis are discussed in the next subsection.

4.4.2 Analysing the Metamodel using Alloy

To perform the analysis, the Alloy Analyser was used to generate an instance of the Alloy metamodel specification. The Analyser requires that a scope is specified and then performs the analysis by exhaustively searching the state space for this

```
-- OCL Specification:
inv noParentsThenNoOverriddenMethods :
  self.parent->isEmpty() implies
    self.overridden_method->isEmpty()
```

```
-- Alloy Specification:
fact noParentsThenNoOverriddenMethods
{
  all c:Class | c.parent = none implies
                c.overridden_method = none
}
```

Figure 4.7: An example of a constraint on the measurement metamodel written in both OCL and Alloy. *This constraint states that if a class does not have any parents then it cannot have any overridden methods.*

scope. We specified a scope of 10 for all elements. The analyser searches for an Alloy instance that contains at most 10 instances of each base class of the metamodel *and* conforms to the well-formedness rules of the metamodel. An instance was produced, thus demonstrating that the well-formedness rules specified for the metamodel were consistent.

We then used the Analyser to search for invalid instances of the metamodel. We specified a scope of 1 for the Alloy specification and manually inspected all instances produced by the Analyser. Each time an invalid instance was found, we added a fact to the Alloy specification to prevent that instance from being generated. For example, we found a metamodel instance where a class could inherit from itself. On completion we had added 11 extra facts to the Alloy specification of the metamodel resulting in a total of 38 facts.

Upon visual inspection of the 38 facts in the Alloy specification of the metamodel, we suspected that a number of these were superfluous. For each of these facts, we converted it into an assertion about the metamodel and then used Alloy to check whether the assertion was valid. If the assertion produced a counterexample then we knew that the constraint was required. If a counterexample could not be found within a reasonable scope then it cannot be guaranteed that the fact is redundant but it can increase our confidence that it is. Therefore, we assumed that the fact was superfluous and omitted it from the specification. During this final anal-

ysis, 25 facts were identified as potentially redundant and removed from the Alloy specification. We also found that a further 2 facts were required to prevent invalid metamodel instances, thus giving us a total of 15 facts in the Alloy version of the metamodel.

4.4.3 Discussion

This approach relies on Jackson's *small scope hypothesis*, which suggests that if a bug exists it will appear in *fairly small* models of a system [Jac06]. So, it is possible our approach may not be applicable to larger metamodels as it may not be feasible to manually inspect all instances of a large metamodel even for a relatively small scope. However, in such a situation it may be possible to apply the approach by partitioning and abstracting the metamodel into the parts that are related to the properties being analysed.

Moreover, we are fully aware that this process is not a completely formalised method for developing and analysing metamodels. However, we believe that this approach gives the developer a formal way of analysing and checking for any suspected deficiencies in their metamodel specification. By iteratively analysing and improving the metamodel, the developer becomes more confident in their specification.

Finally, it is important to note that this approach is not specific to a particular metamodel. It is generally applicable to any MOF-compliant metamodel. In fact, the approach is not restricted to metamodels but is applicable to any kind of model, for example a UML class diagram of a UML model.

4.5 Automatic Test Case Generation for MOF Metamodels

Section 4.3 outlined an approach to the analysis of MOF-compliant metamodels. In this section we describe how this approach was extended to yield an approach to the automatic generation of test data for any software application based or generated from a MOF-metamodel. We then describe an application of the test case generation approach: the construction of a test suite for the automatically generated measurement tool. We use this test suite as input to the measurement tool and use a test oracle to determine whether or not the metric results produced by the tool are

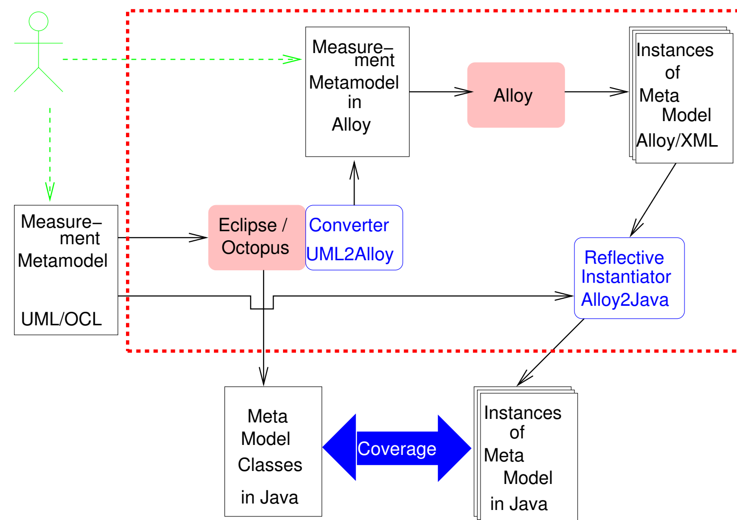

Figure 4.8: Overview of the approach to test case generation for a MOF-compliant metamodel. *The elements in the system are enclosed by a dashed red line. The input, shown on the left, is the metamodel and its constraints expressed using UML and OCL. The outputs, shown at the bottom, are the metamodel implementation and test instances in Java.*

correct. As the test oracle had to be manually constructed it was neccessary that the test suite satisfy the following properties:

1. Each test case should contain a relatively small number of elements.
2. The number of test cases in the test suite should also be relatively small.
3. The test suite should provide as much coverage of the implementation as possible.

4.5.1 Test Case Generation Approach

An overview of the approach is depicted in Figure 4.8. As before, the inputs to the system are the metamodel and its constraints expressed as UML and OCL, and are shown on the left of the figure. The outputs of the system are shown on the bottom, and consist of a Java implementation of the classes and elements of the metamodel and its associated OCL constraints and queries, along with a test suite based on the metamodel. These are linked through a coverage analysis, as described later in this section.

The analysis approach of Section 4.3 was extended with three further steps:

Step 1: Generating a Java implementation of the metamodel. The metamodel and its constraints are depicted using UML/OCL and Octopus is used to generate

Java classes corresponding to the elements of the metamodel. The Octopus tool generates Java classes for each UML class. All attributes and associations are created as fields in the appropriate classes and Octopus creates several accessor and mutator methods for each of these fields. Finally, Octopus creates methods that check the constraints and multiplicities of the model.

Step 2: Generation of test cases using Alloy. The role of Alloy in our system is twofold. First, it allows us to treat the Alloy specification of the metamodel as an Alloy model and check for redundancies or errors in the metamodel specification (see Section 4.4). Second, it allows us to automatically generate valid instances of the metamodel. For this step we created a Java harness to exploit Alloy's model instance finding capabilities. This harness reads an Alloy specification file and uses Alloy to continually generate instances of the Alloy model until all possible model instances have been generated. In our case these model instances correspond to instances of the measurement metamodel. Every Alloy model instance produced during this step is output and stored in XML format for future use.

Step 3: Transformation of metamodel instances to Java objects. One of the central technical contributions of our system is the *Reflective Instantiator*, which transforms the XML versions of Alloy-generated metamodel instances into instances of the Java implementation of the measurement metamodel. The *Reflective Instantiator* parses the XML produced by Alloy and creates instances of the Java classes corresponding to the measurement metamodel elements using the class files generated in step 1. It does this using Java reflection, reading the class names from the XML files and creating instances of these classes. The fields of these classes are set by reading the fields from the XML and calling the appropriate set methods. It is important to note that this process is not tied to any specific metamodel. Since the Alloy specification and Java implementation of the metamodel are generated from the same MOF metamodel, Java reflection can make the link between them without having this information statically hard-coded. Therefore, this program is not specific to the metamodel under consideration and can be used for any metamodel.

Test Group	Alloy Command	No. of Test Cases
1	run show for exactly 1 Type, exactly 1 Attribute, exactly 1 Method, exactly 1 FormalParameter, exactly 1 Invocation	40
2	run show for 1	217
3	run show for exactly 1 ... *all classes listed*	360
4	run show for exactly 2 Type, exactly 2 Attribute, exactly 2 Method, exactly 2 FormalParameter, exactly 2 Invocation	> 528,152

Table 4.3: Groups of generated test cases. *There were four main groups of test cases, generated by varying the settings for Alloys model generator. The number of test cases in each group is shown in the final column.*

4.5.2 Test Case Generation for the Measurement Metamodel

Using our reflective instantiator we were able to automate the generation of a set of test cases for the measurement tool. As we required models with a relatively small number of elements we began by generating Alloy instances using a small scope. Table 4.3 summarises the results of generating these test cases which are partitioned into four different groups:

Group 1 consisted of all possible instances with exactly one instance of each base class in our metamodel.

Group 2 is all possible instances where each base class is observed 0 or 1 times in a metamodel instance.

Group 3 is similar to group 1 except that we defined a scope of exactly 1 for all classes (not just base classes).

Group 4 again is similar to group 1 except that we allowed a scope of exactly 2 for all base classes.

4.5.3 Testing the Implementation of the Metrics

We took the original UML/OCL specification of the metamodel which contained 27 OCL constraints and added the 13 additional constraints discovered during the analysis detailed in Section 4.4. From this updated UML/OCL specification we generated a Java implementation containing 40 constraints in total. All of the test cases summarised in Table 4.3 were used as input to our Reflective Instantiator. For each model, the Instantiator built the instantiation, ran the code to check each of

the 40 OCL constraints, and then systematically deconstructed each model to test the element removal code. As each test model was built it was used as input to the measurement tool and the values for all 44 metrics were recorded.

Since each generated constraint was checked for each test case, this provided further assurance that the reduced set of constraints used to generate the Alloy models instances was sufficient. Further, using such a large number of test cases demonstrates the robustness of the measurement tool and was used as a *smoke test* to ensure that the recorded values were within reasonable boundaries [McC96]. Based on the scope used to generate each of the groups in Table 4.3 we computed the maximum and minimum values possible for each of the metrics. We then identified the models that produced metric values outside of these bounds. The results of this smoke test are discussed later in this section.

Our original intention was to generate a test suite with a relatively small number of test cases whose metric values could be calculated manually, serving as a test oracle for the generated measurement tool. However, since the number of test cases produced is in excess of 500,000, it is necessary to reduce this suite to a more manageable size. We decided to measure the coverage of the implementation in terms of traditional code coverage criteria and to reduce the number of test cases based on these criteria.

Coverage Criteria

Code coverage is a measure of the extent to which the elements of an implementation have been exercised during testing, usually expressed as a percentage. There are several coverage metrics that exist in the literature. We chose to use two well used coverage metrics: line coverage and branch coverage.

- Line Coverage: Also known as statement coverage, this is a measure of how many individual lines of code in an implementation have been executed during testing [ZHM97].
- Branch Coverage: This is a measure of how many conditional points in an implementation have been executed during testing [ZHM97].

Mark Doliner's Cobertura tool was used to measure these two coverage metrics for the metamodel implementation and the implementation corresponding to the metrics. Cobertura is a free Java tool that computes the percentage of code executed

by a set of inputs or test cases [Dol]. It works by instrumenting the bytecode with extra statements that record the lines and branches of the Java implementation that are covered as the code executes.

Cobertura creates a coverage report in HTML or XML format showing the percentage line and branch coverage for each package, class, method and lines of code of the implementation. Using the HTML report it is possible to see exactly which lines of code have not been executed during testing. These lines are highlighted in red. This makes it possible to identify the parts of the implementation that have not been exercised during testing. This information can then be used to create further tests to target the uncovered parts of the implementation.

Coverage Results

It was not possible to achieve full line and branch coverage of the implementation for several reasons, summarised in Table 4.4. Since our test suite only included positive test cases, code that involves catching exceptions when the invariants of the metamodel are violated was not fully covered. Some auxiliary routines, such as alternative set and get methods and constructors were not called in constructing the model. For simplicity, the part of the metamodel dealing with method pointers was not instantiated in Alloy, significantly reducing the number of models created. Thus, excluding these totals from our target coverage gave a maximum possible coverage of 79% for line and 92% for branch coverage.

The results of the coverage analysis is summarised in Table 4.5 on a per-group basis. This table has one row for each of the test case groups described previously in Table 4.3. The data in each case represents the percentage coverage for each of the two coverage criteria. Each row describes the percentage coverage of the measurement metamodel implementation (MM), the metrics implementation (Metrics) and

Reason for exclusion	**Line**	**Branch**
Negative test cases	11%	1%
Extra auxillary methods	7%	3%
Passed-as-Pointer Association	3%	4%
Total excluded	21%	8%

Table 4.4: Line/Branch coverage excluded from the coverage targets. *This table lists three kinds of code excluded from the coverage targets, along with the percentage of lines/branches for each kind.*

Test Group	Cum. Line Coverage			Cum. Branch Coverage		
	MM	Metrics	All	MM	Metrics	All
1	45%	75%	54%	56%	66%	59%
2	50%	75%	57%	63%	66%	64%
3	50%	75%	57%	63%	66%	64%
4	70%	99%	79%	89%	99%	92%

Table 4.5: A breakdown of the metamodel coverage for each of the test groups in Table 4.3. *The numbers presented for each group represent the cumulative coverage achieved, including the previous test groups.*

the combined percentage coverage (All). Furthermore, each row represents *cumulative* coverage; for example, the line coverage value of 57% for group 2 includes the 54% line coverage achieved by group 1. As can be seen from Table 4.5, the smaller test suites exhibit relatively poor coverage.

Test Oracle Construction

In this subsection we consider the construction of a *reduced* test suite that achieves the maximum coverage criteria possible for use in constructing a test oracle for the measurement tool.

A number of techniques exist that can reduce test suites based on various constraints. For example, Harrold *et al.* outline techniques for test suite reduction and prioritisation based on coverage criteria [HGS93]. However, since our test cases were being generated by Alloy roughly in order of size, a simpler approach was taken to test suite reduction:

1. As each test case is executed, the cumulative coverage of both criteria is recorded.
2. Any test case that causes an increase in any one of the two coverage figures is added to the reduced suite.
3. This process is continued until either the maximum coverage has been achieved for both criteria or until all test cases have been examined.

In general this process will not perform as well as that of Harrold *et al.*, but it is much simpler to implement. Applying this technique to the test cases, we generated a reduced test suite of 14 unique test cases. Table 4.6 lists the cumulative coverage data for each of these cases, labeled T1-T14. Three of these cases (T1-T3) originated from group 1, three (T4-T6) from group 2, and eight (T7-T14) from group 4. It may appear that some of these test cases do not cause an increase in

Test Case	Cum. Line Coverage MM	Metrics	All	Cum. Branch Coverage MM	Metrics	All
T1	40%	69%	49%	50%	58%	52%
T2	45%	75%	54%	56%	66%	59%
T3	45%	75%	54%	56%	66%	59%
T4	49%	75%	57%	62%	66%	64%
T5	49%	75%	57%	63%	66%	64%
T6	50%	75%	57%	63%	66%	64%
T7	66%	94%	75%	85%	93%	87%
T8	66%	94%	75%	85%	93%	87%
T9	67%	95%	75%	89%	94%	88%
T10	69%	95%	77%	89%	94%	90%
T11	69%	95%	77%	89%	94%	90%
T12	70%	98%	78%	89%	98%	92%
T13	70%	99%	79%	89%	99%	92%
T14	70%	99%	79%	89%	99%	92%

Table 4.6: Test cases in the reduced test suite. *This table lists the 14 test cases in the reduced suite, along with the cumulative coverage figures for the two coverage criteria.*

coverage figures, for example T3, T10 and T12. This is because the coverage figures in Table 4.6 have been rounded to the nearest whole number and any increases in coverage by a fraction of a percentage are not be reflected.

The 14 test cases almost achieved the maximum coverage possible. By inspecting the HTML report from the Cobertura tool we were able to identify 9 lines of code that had not been covered by the reduced test suite. On further analysis of these lines of code we identified them as belonging to the ICP and NIH_ICP metrics in the Coupling metrics set. To cover these remaining 9 lines of code required a model with the following properties:

- a class with a method that polymorphically invokes one of its ancestor's methods
- a class with a method that polymorphically invokes at least one method that is not in the set of new or overridden methods of that class

We then used Alloy to generate a valid metamodel instance to cover these situations. This was achieved by adding a fact to the measurement metamodel that stated that the above properties must hold true for the metamodel. This model was added to our test suite and increased the coverage to the maximum value possible of 79% for code coverage and 92% for branch coverage.

The 15 test cases were then used to manually create a test oracle for the measurement tool. All 44 metrics were calculated by hand and recorded for each of the 15 test cases. The results of these metric calculations for each of the 15 models can be found in the supplementary material accompanying this thesis. We compared these values with the actual values computed by the measurement tool. In the next subsection, we briefly discuss the results of this along with the results from the smoke test.

4.5.4 Discussion

Using the above procedure we uncovered 6 bugs in the measurement tool. Four of these were detected by the smoke test and 2 with the test oracle. During the smoke test, it was discovered that several of the metric definitions (Co, NewCo, LCOM5, COF, TCC and LCC) do not take into account situations that result in a division by 0. This resulted in the implementation of these definitions either throwing an exception or reporting the value as `NaN` indicating the result is not a number or as either `+/-Inf` indicating the result as plus or minus infinity. Rather than modify the original definitions we chose to record the invalid value that was computed for the metric and leave the decision to the user on how to interpret it. This demonstrates an important feature of the measurement approach, it allows metric definitions to be easily implemented and tested which helps to identify shortcomings with the definitions before they are presented in the literature. A bug was also found that resulted in incorrect values being computed for two of the cohesion metrics. The metrics LCOM1 and LCOM2 both use the same OCL auxiliary operation to compute the set of method pairs in a `Class`. It was discovered that the OCL operation was counting each method pair twice and thus resulted in values outside of the expected bounds for the metrics. This error was corrected at the OCL level. A coupling metric, COF and a cohesion metric Co also produced values outside of the expected bounds for these metrics. The cause of these two bugs were identified as misplaced brackets in the generated code corresponding to the metrics. This bug was corrected at the OCL level.

The two bugs uncovered using the test oracle were found in the cohesion metrics, NewCo, LCOM3 and LCOM4. These bugs were traced back to errors in the OCL definition of the metrics i.e. the metric definitions had been incorrectly specified in OCL. Once all the bugs in the OCL were corrected, the measurement tool

was regenerated. The measurement tool was tested again using the smoke test and test oracle.

In summary, we were able to partition the types of errors we found into three categories. The first category are bugs that are a result of the metric definitions themselves. For, example when a metric has no provision for a division by 0. The second category, are those introduced in the OCL where the definition has been incorrectly specified, for example an omission in the OCL definition. The last category consists of errors introduced by Octopus in transforming the UML/OCL to Java, for example a misplaced bracket. Overall, our experience found this to be a relatively simple and effective way of increasing our confidence in the correctness of the automatically generated measurement tool.

4.6 Summary

In this chapter we presented an approach to analysing MOF-compliant metamodels. We also presented a measurement metamodel for coupling and cohesion metrics based on the work of Briand *et al.* and described how we used our approach to construct and analyse the metamodel. The metamodel and well-formedness rules were expressed in UML and OCL and a Java implementation and Alloy specification of the metamodel were generated by third-party tools. We used the Alloy specification to examine and validate the metamodel constraints, and to generate instantiations of the metamodel.

We defined a set of existing coupling and cohesion metrics using the measurement metamodel and used dMML to automatically generate a measurement tool for these metrics. We used Alloy to automatically generate all possible instances of the measurement metamodel for a relatively small scope. We implemented a reflective instantiator to transform these Alloy generated models into an instantiation of the Java implementation of the metamodel, thus creating a test suite for the automatically generated measurement tool. Finally, we evaluated the adequacy of the test suite using traditional code coverage criteria.

We identify the principal contributions of this chapter as:

- An **approach to the analysis** of MOF-compliant metamodels that is supported by an integrated tool framework.
- The **development and analysis** of a MOF-compliant metamodel for coupling

and cohesion metrics, based on the work of Briand *et al.*, and the elimination of redundant constraints in that metamodel.

- The **automation** of the generation of metamodel instances from a UML/OCL specification that can be used as test data for metamodel-based software.
- A **coverage-based analysis** of the Alloy-generated test suite in terms of traditional code coverage, thus “completing the circle” between lightweight formal methods and standard software testing techniques.

Chapter 5

Testing Model Transformations for the Measurement Metamodel

This chapter completes the measurement approach by introducing support for model transformations and transformation testing.

5.1 Introduction

The measurement approach presented in this thesis is centered on developing a set of model transformations from one modelling language, such as UML or Java, into the measurement metamodel. Hence, in order to ensure the correctness of the resulting metrics it is important that the model transformations are correct, that is the target models produced by the transformation implementation are correct with respect to the model transformation specification [Lam07]. The work in the previous chapter has dealt with ensuring the correctness of metamodels and in particular the measurement metamodel; in this chapter we are concerned with testing and validating model transformations.

One important aspect of software testing is determining the degree to which the test cases used in testing exercise the system under test [Bin00]. Since metamodels can be implemented (often automatically) in program code, applying coverage measures to this generated code provides one means of measuring metamodel coverage. This approach was taken in the previous chapter where line and branch coverage were used to evaluate coverage for an implementation generated by the Octopus

tool [MP08b]. While this approach has the advantage of simplicity, it is rather indirect, and depends to some degree on decisions made by the code generator.

A number of coverage measures for class diagrams are defined by Andrews *et al.* [AFGC03] and used by Fleurey *et al.* to develop a test suite for a model transformation [FSB04]. The advantage of this approach is that coverage is calculated directly on the metamodel; a disadvantage is that the role of coverage criteria at the UML level is not as mature a field as coverage at the programming level. Both of the above approaches, using program and UML based measures, rely on generating a test suite that adequately covers the input domain of a transformation, as defined by the input metamodel, or a relevant subset. However, there is little work on directly considering the coverage of the transformations themselves.

In this chapter, we describe a generally applicable approach to testing model transformations where the target model is a measurement metamodel instance. We also present details of two model transformations, one that converts UML class diagrams into instances of the measurement metamodel described in Chapter 4 and one that converts Java programs into instances of the measurement metamodel. We address the problem of transformation testing by focusing on developing a set of coverage measures for model transformations and report on our experience of applying this approach to our model transformations.

5.2 Model Transformation Languages

In this section we describe the development of the final part of the measurement approach which incorporates a model transformation language into the approach and provides tool support to perform the transformations. Many languages and tools have been proposed to specify and execute model transformations, each of which take a different approach to transforming the source model to the target model. In order to choose a transformation language to incorporate into the measurement approach we set out a list of requirements for the transformation language and tool based on the important characteristics of transformation languages outlined by Mens *et al.* [MG06]. These requirements include being compliant with the relevant standards including MOF, EMF and QVT and having support for fully automating the model transformation process. In total, 17 languages and tools were evaluated according to the our criteria and the results of this evaluation are summarised in Table B.1 of Appendix B.

After the examination of these transformation languages we made the pragmatic decision to choose the Atlas Transformation Language (ATL) [JK05, ATL] as it was one of only a few languages that satisfied all the requirements that were set out. Also, from the set of languages satisfying all the criteria, ATL is one of the longest established transformation languages, has a large user community and has a rich selection of sample ATL transformations to avail of which are stored as a collection of transformations referred to as a transformation zoo [Ecla].

5.2.1 The Atlas Transformation Language

The ATL has been developed by the ATLAS INRIA and LINA research group in response to the OMG MOF/QVT RFP [OMG02]. ATL is not fully QVT compliant but does support a large number of the QVT requirements such as compatibility with MOF, XMI, and the use of the OCL for navigation and as such it is is referred to as a QVT-like language. It is a hybrid transformation language which means that it provides both declarative and imperative constructs for writing transformations. Users write the transformations by defining rules that specify how elements in the source model are transformed into elements in the target model by referencing elements in both the source and target metamodels. ATL is not just a model transformation language but is supported by a set of development tools including an editor, a compiler and a virtual machine, all of which are supplied as an Eclipse plug-in. To perform a transformation written in ATL a user supplies the transformation definition, the source and target metamodels in MOF or Ecore format. The ATL engine performs the transformation to produce the target models [JK05, JAB+06, ATL].

An example of an ATL model transformation is given in Figure 5.1. This is a simplified version of the *FamiliesToPersons* transformation from the ATL website [ATL]. The aim of this transformation is to convert a list of families into a list of people. The source and target models of this transformation are shown in Figure 5.2. The source metamodel, shown on the left of the figure models a family, consisting of members that have a first name. The target metamodel shown on the right of the figure models a `Person` as having a name and being either male or female.

The ATL transformation definition is composed of three parts, a header, the helpers and the rules. The header declares the name of the transformation and declares variables for the source and target models (”IN” and ”OUT”) and the metamodels of the source and target models (”Persons” and ”Families”). Helpers are

```
module Families2Persons;
create OUT : Persons from IN : Families;

helper context Families!Member def: isFemale() : Boolean =
       if not self.familyMother.oclIsUndefined() then
              true
       else
              if not self.familyDaughter.oclIsUndefined() then
                     true
              else
                     false
              endif
       endif;

      rule Member2Female {
from
   s : Families!Member (s.isFemale())
to
   t : Persons!Female (
       fullName <- s.firstName
)
}
```

Figure 5.1: The sample *FamiliesToPersons* ATL transformation. *This piece of ATL code shows a sample transformation that convert a list of families into a list of people and consists of a transformation rule and a helper operation.*

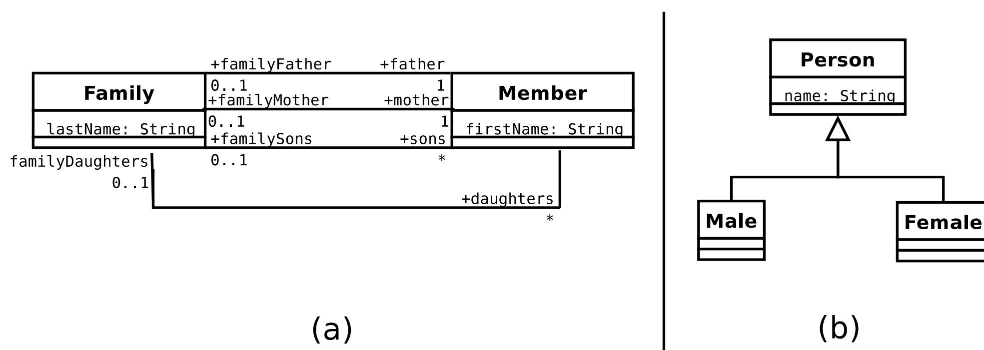

Figure 5.2: The source and target metamodels of the *FamiliesToPersons* Transformation. *This figure shows the metamodel Families used as the source metamodel in the transformation and the metamodel Persons used as the target metamodel of the transformation.*

auxiliary functions that return information about the source model and their main purpose is to facilitate code re-use. The *FamiliesToPersons* transformation contains

a single helper, `isFemale()` which queries a `Member` instance in the source model and returns true if it plays the role of `mother` or `daughter` in a relationship with a `Family` instance.

Rules are core to ATL transformations as they describe how elements of the target model are created from elements of the source model. In this example, `Member2Female`, specifies that a `Female` instance in the `OUT` model is to be created for every `Member` instance in the `IN` model where the `isFemale()` helper returns true for that `Member` instance. The rule also specifies that the `fullname` of the created instance is set to the `firstName` of the `Member` instance.

5.2.2 Integrating ATL with dMML

Figure 5.3 presents an overview of our final measurement approach which incorporates the ATL model transformation language. The inputs to this system, shown on the left of Figure 5.3 are a set of source models written using a modelling language, and the output, shown on the right, is a set of metrics for those models.

Each source model, for example a UML class diagram or Java program is used as input to the ATL tool in Ecore format as an instance of its language metamodel. This ATL tool has been adapted to run in stand-alone configuration (i.e. from the command-line) to facilitate the iteration and automation of the process and to allow metric calculations to be executed in batch. The source models, presented as instances of the language metamodel, are transformed using a model transformation,

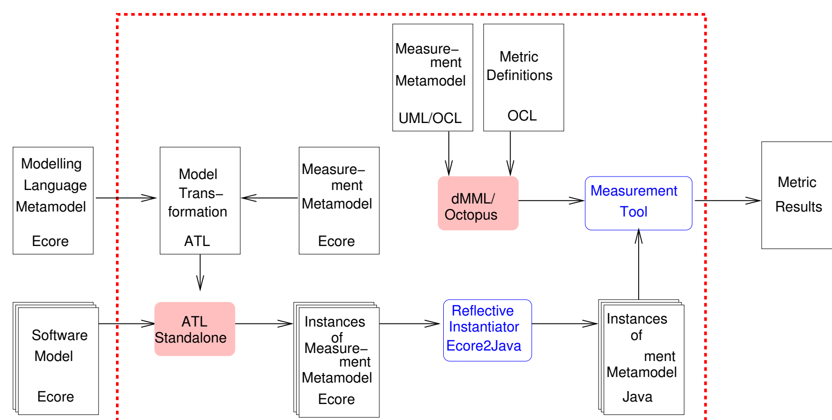

Figure 5.3: Overview of the aproach used to calculate metrics for class diagrams. *The input to the system is a UML class diagram, and the outputs are the metrics for the UML model. The system itself is delineated by a dashed box.*

written in ATL, to instances of the measurement metamodel in Ecore format. Recall that for dMML to work for other languages, in this case Ecore, a tool must be constructed to convert instances of the metamodel in Ecore format to Java instances for import into the automatically generated measurement tool. This tool has been constructed as a reflective instantiator by reading the class names and attributes from the Ecore/XMI file and using Java reflection to create instances of these models. These Java instances of the measurement metamodel are used as input to the tool that calculates the metrics. The metrics themselves have been specified as OCL queries over the measurement metamodel, and the tool to calculate them has been automatically generated as a Java program.

The important point to note here is that the approach shown in Figure 5.3 is designed to be reusable for different source metamodels, so that metric calculations can be performed using the same definitions. To calculate metrics for any model, such as a UML class or sequence diagram or program code, the only requirement is to be able to construct an ATL model transformation to plug in to the approach.

We have used this approach to calculate metrics for UML class diagrams and Java programs by writing transformations to convert class diagrams and Java programs to instances of the measurement metamodel. The *UMLClassDiag2Measurement* transformation and the *Java2Measurement* transformation are presented in the following sections.

5.3 Measurement of UML Class Diagrams: The *UMLClassDiag2Measurement* Transformation

In Chapter 2, we introduced the basic concepts of model transformations. In this section we describe the formulation of the *UMLClassDiag2Measurement* transformation from the UML metamodel to the measurement metamodel.

5.3.1 Source and Target Metamodels

Source Metamodel

The source metamodel in the transformation is the UML 2 metamodel [OMG07b]. This metamodel specifies the constructs that may be used in a UML model and the relationships between these constructs. For example, the part of the UML meta-

model that is specific to class diagrams defines the concepts of class, property and operation and states that a class includes properties and operations. The implementation of the UML 2 metamodel provided by the Eclipse UML2 project was chosen [Ecle].

Target Metamodel

The target metamodel in the transformation is the measurement metamodel that has been presented in Chapter 4.

5.3.2 The Transformation Rules

The *UMLClassDiag2Measurement* transformation was written in 230 lines of ATL code composed of 10 transformation rules and 22 helpers. A number of the transformation rules are reasonably straightforward to specify, since the source and target metamodels have a number of elements in common. For example, classes and generalisations in a UML class diagram are mapped to class elements and parent/child relationships in the measurement metamodel. The UML metamodel contains a number of “similar” elements. However, these elements do not exist as distinct elements in the measurement metamodel so the decision was taken to map them to the same element of the measurement metamodel. For example, association classes and interfaces are both mapped to classes in the measurement metamodel. In some cases, elements are slightly re-interpreted; for example UML enumerations and data-types are mapped to user-defined types in the measurement metamodel, and UML primitive types correspond to “built-in” types in the measurement metamodel.

However, the measurement metamodel also makes distinctions that are not explicitly present in UML class diagrams. For example, the measurement metamodel requires that the methods in a class be categorised as constructor, destructor, accessor, mutator or general methods, since the kind of method can have an impact on coupling and cohesion metrics. The approach taken in the *UMLClassDiag2Measurement* transformation was to identify and categorise such methods syntactically, based on standard naming conventions. For example, if a method has a name that begins with “get” followed by the name of an attribute of a class and it has a return parameter that is the same type as that attribute it is categorised as an accessor.

The measurement metamodel also requires that we introduce a distinction between methods based on whether they were new, inherited or overridden. Thus *overridden* operations are identified in the UML context as operations that belong to a class which has an ancestor with an accessible operation with the same name and signature. Similarly, *inherited* operations are selected from the inherited members of the class and any operations that are identified as overridden are removed from this set. All other operations are identified as *new* operations. The declared and implemented methods of a class can be determined from the new, overridden and inherited methods.

While writing the transformation it was decided to make use of derived attributes as they are already implemented in the UML2 metamodel. While it would be possible to re-implement this functionality in the ATL transformation, it seemed logical to reuse definitions that were already part of the target metamodel. From a testing point of view this could be problematic, since if we find an error in the model transformation it could be either in our ATL code or in the computation of derived attributes in the UML metamodel. For our purposes we will assume that the UML metamodel implementation is correct and make use of derived attributes in the transformation rather than write more complicated transformations using only non-derived attributes.

The final decision that was required was regarding mapping UML associations and other UML relationship types such as dependency, usage, abstraction and substitution, since there are no corresponding concepts in the measurement metamodel. We made the decision to map association ends of associations to attributes in the measurement metamodel if and only if these ends are navigable. The owner of the attribute is the class on the opposite end of the association. All other relationships are ignored as we cannot guarantee that the different relationships mean that there will be an attribute created in the owning classes.

Any properties that belong to a class are also mapped to attributes in the measurement metamodel. An example of this transformation rule `FieldProperty2-Attribute` is given in Figure 5.4. This rule specifies that a property is transformed into an attribute if that property belongs to a class rather than an association. This rule means that for each property in the source model an attribute is created in the target model provided the guard of the rule evaluates to true. The guard is defined using the helper function `isField`. This helper returns true if the owner of the property is a class, interface or association class. In the case of the property

```
rule FieldProperty2Attribute {
from
  p : UML!Property(p.isField())
to
  a : MeasurementMetamodel!Attribute(
    name <- p.name,
    type <- p.type
  )
}

-- Returns true if the property is a field in a Class, Interface
-- or AssociationClass, and returns false otherwise
helper context UML!Property def: isField() : Boolean =
  if self.owner.oclIsTypeOf(UML!Class) then
    true
  else
    if self.owner.oclIsTypeOf(UML!Interface) then
      true
    else
      if self.owner.oclIsTypeOf(UML!AssociationClass) then
        if self.owningAssociation.oclIsUndefined() then
          true
        else
          false
        endif
      else
        false
      endif
    endif
  endif;
```

Figure 5.4: Example of a transformation rule in ATL. *This piece of ATL code shows a transformation rule, `FieldProperty2Attribute`, and an associated helper operation called `isField`.*

being owned by an association class an additional check is performed to ensure that the property is not an association end. In all other situations the helper returns false. The name and type of the newly created attribute is set using the name and type of the attributes matching property in the source model.

5.4 Measurement of Java programs: The *Java2Measurement* Transformation

In this section we describe the formulation of the *Java2Measurement* transformation from the Java programs to the measurement metamodel.

5.4.1 Source and Target Metamodels

Source Metamodel

The source metamodel in the transformation is a metamodel for the Java language. Since our transformations are based on EMF metamodels we needed an EMF metamodel for Java. We choose to use the Java metamodel provided by the SpoonEMF project [ND08]. Spoon is a project that provides a core API and tool support for static analysis and generative programming in Java 5 and 6. Spoon-EMF provides an EMF compliant metamodel equivalent to the Spoon API and can be used to convert a complete Java implementation into a single XMI file containing an instance of the metamodel [ND08, MJCH08].

Target Metamodel

Again, the target metamodel in the transformation is the measurement metamodel presented in Appendix A.

5.4.2 The Transformation Rules

The *Java2Measurement* transformation was written in 249 lines of ATL code composed of 11 transformation rules and 29 helpers. Again several of the transformation rules are reasonably straightforward to specify, since the source and target metamodels have a number of elements in common. Classes and interfaces in the Java metamodel are mapped to classes in the measurement metamodel. Elements of type “built-in” are created in the measurement metamodel from `TypeReference` elements of the Java metamodel where the name of the `TypeReference` is one of Java’s built in types e.g. `float`, `int` or the `String` class.

The child/parent relationship for a class in the measurement metamodel is resolved through the `superclass` and `superinterface` relationships of that classes corresponding element in the Java metamodel. The `superclass` and

superinterface relationships resolve to TypeReference elements of the Java metamodel. The TypeReference element does not have a link to the Type element (e.g. class or interface) in the Java model that it references but only stores the name of the Type. These superclasses and superinterfaces of the class need to be resolved by looking at all classes and interfaces in the Java model and choosing the class or interface that has the same name as the name specified in the TypeReference element.

Two rules are required for transforming a Field in the Java metamodel to an Attribute of the measurement metamodel. The first one is for attributes or fields that have a type that resolves to "built-in" type and the other is for those attributes that have a type that resolves to something other than a "built-in" type. The type is resolved through the TypeReference associated with the Field element. Again the TypeReference element does not have a link to the Type element in the Java model that it references but only stores the name of the Type. Therefore, the type of the attribute is resolved by examining all the classes and interfaces or all the primitive types in the Java model and choosing the class, interface or primitive type that has the same name as the name specified in the TypeReference element. Finally, the rules for determining the methods that reference the field or attribute are the same in both cases. This is done by examining all FieldAccess elements in the Java model and choosing those that access the field under consideration. Then for each of these FieldAccess elements, the method in which it occurs is resolved by traversing up the parent hierarchy until the owning method is reached.

Although the Java metamodel makes distinctions between constructors and methods, both these elements are mapped to methods in the measurement metamodel. Figure 5.5 shows the transformation rule used to convert constructors of the Java metamodel to methods of the measurement metamodel along with two helpers required by the transformation rule. The constructor element in the Java metamodel does not have a name attribute but this can be derived from the name of the class that owns the constructor. The isAbstract and isPublic attributes of the method are set using the isAbstract() and isPublic() helper functions. These two functions check the Modifiers association of the constructor to determine if it includes the abstract or public enumeration literals. The type of the method created in the measurement metamodel is set to be a constructor. The type of all other methods created in the measurement metamodel are identified syntactically, simi-

```
rule Constructor2Method {
from
  javac : Java!CtConstructor
to
  mm : Measurement!Method(
    name <- javac.Parent.SimpleName,
    param <- javac.Parameters,
    isAbstract <- javac.isAbstract(),
    isPublic <- javac.isPublic(),
    type <- # constructor
)
}

-- Determines if the method in the Java model is public
helper context Java!CtExecutable def: isPublic() : Boolean =
    self.Modifiers->includes(# public);

-- Determines if the method in the Java model is abstract
helper context Java!CtExecutable def: isAbstract() : Boolean =
    self.Modifiers->includes(# abst);
```

Figure 5.5: Example of a transformation rule in ATL. *This piece of ATL code shows a transformation rule, `Constructor2Method`, and two associated helper operations called `isAbstract` and `isPublic`.*

lar to how we categorised methods in the *UML2ClassDiagMeasurement* transformation in Section 5.3. Methods are also categorised according to new, inherited, overridden, declared and implemented using the same rules as those outlined in the *UMLClassDiag2Measurement transformation*. Parameters are mapped to formal parameters in the measurement metamodel and their type resolved using a similar approach to how the type of an attribute is resolved.

Elements that were not present in the *UMLClassDiag2Measurement* transformation are method invocations and calls to constructors. Invocations in the measurement metamodel map to invocations in the measurement metamodel only if the method being called is present in the Java model, thus ensuring the callee of the invocation can be resolved. The callee of the invocation is identified from the `ExecutableReference` element of the invocation. However, as the `Executable-Reference` has no link to the `Executable` element (i.e. method or constructor) that is being referenced or called, these `Executable` elements (methods and constructors) need to be resolved by looking at all methods and constructors in the Java

model and choosing the correct one based on criteria such as name and parameter list which is specified in the `ExecutableReference` element. The caller of the invocation is identified by traversing up the parent hierarchy of the invocation to identify the method that the invoction belongs to. The type of the invocation is determined from the `Static` attribute of the `ExecutableReference` element. Calls to constructors are represented by their own entity `NewClass` and are only transformed to an invocation if they appear inside a method.

5.5 Transformation Testing

As described in the previous two sections, writing the ATL transformations involved making a number of distinctions between elements in the source model and the target model, leaving considerable scope for error. Therefore, we tested our transformations on a number of UML class diagrams and Java programs and, in this section, we describe the approach used to test the transformations and discuss the coverage measures used to ensure that the test suites were adequate.

5.5.1 ATL Coverage Criteria

An ATL transformation includes rules to match elements in the source model as well as instructions to build the elements in the target model. By analogy with a standard grammar-based model such as that used in a yacc/bison file, the ATL transformations include both the rules to match the input and actions to build the output. Based on this analogy, we posit that concepts of coverage in an ATL transformation should include rule coverage for the matching part, and traditional code coverage for the generation part. Thus we define:

Rule Coverage. This is analogous to rule coverage in a grammar [Pur72]: it is simply the percentage of rules that were executed at least once during a transformation.

Instruction Coverage. This is analogous to line coverage in a high-level language [Bei90], with the additional benefit that formatting and layout do not effect the totals. The instruction coverage for a set of transformations is the percentage of instructions that were executed at least once during the transformation.

Branch Coverage. This measures, for each branch in a program, whether the true and false paths were taken. Thus each branch contributes a figure of 0, 1 or 2 to the total coverage, depending on whether neither branch, just one branch or both branches were taken at any stage during the transformation.

In order to calculate the rule, instruction and branch coverage of an ATL transformation it is necessary to profile the operation of each ATL rule as the transformation takes place. Fortunately the design of the ATL system provides two useful features that facilitate this. First, the ATL rules are actually executed on top of a special-purpose virtual machine [JA06]. Second, it is possible to run the ATL system in debug mode which prints out a step-by-step execution log of instructions on this virtual machine.

The ATL virtual machine is similar in concept to the Java Virtual Machine (JVM) which greatly eases comprehension. It has instructions to access and create model elements, to manipulate data on the stack, and control instructions for selection, iteration and method calls. Thus, to measure coverage of ATL transformations we have implemented a program or *Coverage Analyser* that works in two phases:

1. First, we process the file of compiled ATL instructions (represented in XML format) to extract information about the rules or operations, instructions and branch locations and targets

2. Second, we run the transformation and process the resulting log file to record the actual coverage data for that transformation.

We now describe how the three coverage criteria are measured from the log file. Each transformation rule is represented as an operation in the compiled ATL file and is executed on the ATL virtual machine. Therefore, implementing rule coverage involved tracking and recording the calls to the operation corresponding to each rule. The log file produced by running the transformation lists each instruction as it is executed, so it is relatively straightforward to measure this coverage as each instruction corresponds to a line in the ATL transformation file. Branches in an ATL transformation are represented by IF and ITERATE instructions in the compiled ATL file, and whether they evaluated to true or false can be determined from the log file. For the ITERATE instruction, evaluation to true or false corresponds to iterating over an empty or non-empty collection respectively.

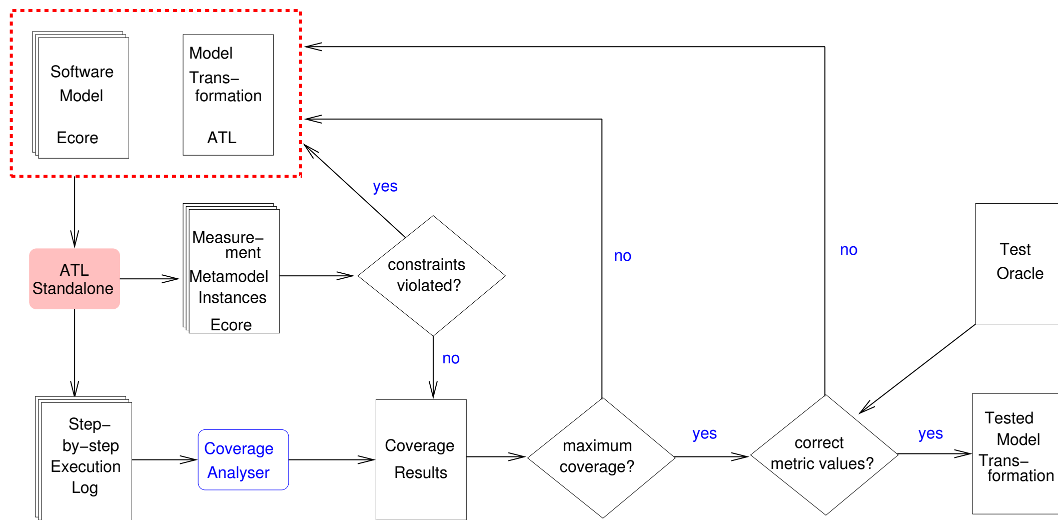

Figure 5.6: Overview of the approach used to test our model transformation. *The input to the approach is the set of source models and the model transformation and the output is a tested model transformation.*

These coverage criteria can then be used to evaluate the adequacy of any test suite used when testing model transformations. Since our measurement approach is concerned with transforming software models to instances of the measurement metamodel, in the next section we outline a strategy for testing model transformations where the target model is a measurement metamodel instance and use the coverage criteria outlined here to ensure that the test suite used is adequate.

5.5.2 Testing Strategy

An overview of the strategy to testing model transformations is depicted in Figure 5.6. Each source model, in our case a UML class diagram or Java program, is used as input to the standalone ATL tool. A successful transformation produces two outputs, an output model which is an instance of the measurement metamodel in Ecore format and an execution log which is processed to extract coverage information.

For the transformation to be correct the following two criteria must be satisfied:

1. The output model should conform to the measurement metamodel.
2. The calculated metrics for the output model should be the same as the metrics for the source model.

Testing the model transformation is a three step process:

Step 1: Checking Metamodel Conformance:. After a set of input models or test suite are put through the transformation the output models that are produced are checked to ensure they are well-formed according to the OCL constraints for the measurement metamodel. If any constraints are violated then these constraints are inspected and the information is used to identify errors in the transformation. These errors are then corrected and the input models are put through the corrected transformation. This process is repeated until all output models conform to the measurement metamodel.

Step 2: Coverage Analysis:. When all output models conform to the measurement metamodel constraints the next step is to examine the transformation coverage. The execution log produced during the transformation execution is used as input to our *Coverage Analyser* tool which determines and records the Rule, Instruction and Branch coverage for the test-suite as described earlier in this section. If full coverage has not been achieved then the output from the coverage analyser is used to determine the parts of the transformation that have not been exercised and this information is used to create further source models to cover these items. These models are added to the test suite and the testing process returns to Step 1. However, if full coverage has been achieved then the second correctness criteria can now be checked; the metric values produced for the output models must be the same as those for the input models.

Step 3: Test Oracle Construction:. This step involves the construction of a test oracle for the measurement tool. The test oracle consists of a set of known metric values for each of the input models in the test suite. This test oracle is constructed by calculating and recording the correct metric values for each input model. The results produced by the measurement tool for the output models are then compared with the correct results for the equivalent input model. At present this is all done manually; this is feasible as long as the input models, in this case the class diagrams and Java programs are kept to a reasonable size.

5.6 Testing the Model Transformations

In this section we present the results of applying the testing strategy presented in the previous section and depicted in Figure 5.6 to the *UMLClassDiag2Measurement*

the *Java2Measurement* transformations.

5.6.1 Testing the *UMLClassDiag2Measurement* Transformation

Test Suite

In order to test the transformation we first had to construct a test suite. To achieve this we gathered a set of UML class diagrams to use as a set of test cases. The test cases used in our study were taken from the Eclipse UML2 Tools project, an Eclipse project that aims to provide a set graphical editors that can be used to view and edit UML2 models [Eclf]. To demonstrate the capabilities of the UML2 Tools project 20 sample class diagrams from the UML 2.0 metamodel specification [OMG07b] have been implemented using UML2 Tools. These 20 diagrams exercise the various features of the class diagram part of the UML metamodel. Each of these class diagrams is described briefly in Table 5.1, mainly to provide reference to the original source. Each test case has a description which corresponds to the name of the class diagram in the UML2 Tools project and also to the figure number and caption of the diagram as it appears in the UML specification [OMG07b]. While there was no coverage data or analysis provided with these models, they were selected as they presumably covered a sufficient range of features in class diagrams. Using this test suite we applied the testing strategy outlined in the previous section and the results of this are presented in the remainder of this section.

Checking Metamodel Conformance

After the original set of test cases were put through the transformation, 14 out of the 20 target models that were produced violated the constraints or well-formedness rules of the measurement metamodel. In total 53 constraints were violated, but on further inspection it was found that only 5 of these constraints were unique. We traced these violations back to four errors, two omissions in the measurement metamodel and two in the model transformation.

The two omissions in the measurement metamodel were due to an over-constraint: we had required that attributes and parameters in the measurement metamodel should have a corresponding type, but this is optional in the UML metamodel. To correct this problem we modified the measurement metamodel to allow `Attributes` and `FormalParameters` to have an optional type by changing the multiplicity from

Test Case	Description used in UML2 Tools project
1	7.19 Graphic notation indicating exactly one association end owned by the association
2	7.20 Combining line path graphics
3	7.21 Binary and ternary associations
4	7.22 Association ends with various adornments
5	7.23 Examples of navigable ends
6	7.24 Example of attribute notation for navigable end owned by an end class
7	7.25 Derived supersets (union)
8	7.26 Composite aggregation is depicted as a black diamond
9	7.27 An Association Class is depicted by an association symbol (a line) and a class symbol (a box)
10	7.28 Class notation - details suppressed, analysis-level details, implementation-level details
11	7.30 Examples of attributes
12	7.32 Comment notation
13	7.33 Constraint attached to an attribute
14	7.40 Example of element import with aliasing
15	7.48 Multiple ways of dividing subtypes (generalization sets) and constraint examples
16	Example of stereotyped class notation
17	7.27 Association Class is depicted by a line and a box
18	7.54 Instance specifications representing two objects connected by a link
19	Figure 7.39 Example of element import
20	Figure 17.19 Template Class and Bound Class

Table 5.1: A summary of the test cases. *This table lists the class diagrams from the UML2 Tools project [Eclf] (release 0.8.0 of 11 June 2008). In future tables we refer to these models by number only; this table can be used to refer back to models in the UML2 Tools distribution and UML 2 superstructure specification [OMG07b].*

`1` to `0..1`. The final specification of the measurement metamodel in Appendix A has been updated to incorporate this change. The remaining two errors were found in the model transformation, one as a result of an omission in the transformation. Navigable association ends in the UML class diagrams were being mapped to attributes in the measurement metamodel, but no owning class was being identified for them which is mandatory in the measurement metamodel. We corrected this by setting these attributes' owning class to be the class on the opposite end of the association. The other error was as a result of the incorrect identification of attributes

for the ancestors of a class and thus the set of declared attributes of that class was also incorrect.

Coverage Analysis

The results of the coverage analysis are summarised in Table 5.2 on a per-model basis. This table has one row for each of the UML models described previously in Table 5.1. The data in each row represents the percentage coverage when the corresponding UML model was transformed into an instance of the measurement metamodel using the *UMLClassDiag2Measurement* transformation. The final row of Table 5.2 shows the cumulative total coverage when all 20 UML models were transformed using the *UMLClassDiag2Measurement* transformation, and thus represents the total coverage for all models considered as a test suite.

In total the *UMLClassDiag2Measurement* transformation contained 10 ATL rules which, along with the helper operations, were implemented in 1686 virtual machine instructions, of which 103 were IF or ITERATE instructions, giving a total of 206 possible true/false branches. Thus, for example, we can tell from Table 5.2 that UML model No. 5 covered 3 rules, which caused the execution of roughly 614 instructions and 48 of the possible true/false branches.

Overall we can see from Table 5.2 that each individual UML model covers between 10% and 50% of the rules, between 29% and 62% of the instructions and between 20% and 39% of the branches. While the spread of values is quite small, we can see a tendency for rule, instruction and branch coverage to increase in tandem.

The total coverage data, shown in the last row of Table 5.2, provides a measure of the effectiveness of the set of 20 UML models considered as a test suite. As can be seen the cumulative coverage is quite high at 80% rule coverage and 76% instruction coverage, though somewhat lower for branch coverage at 51%. Given that the input UML models were not designed with coverage in mind, this is a reasonably satisfactory starting point for developing a more comprehensive test suite.

Further analysis of the low branch coverage data from Table 5.2 is given in Table 5.3. From this table we can see that 24% of the missing coverage is due to branch instructions not being executed at all, which is in line with the overall instruction coverage figure. A further 18% of branch instructions were only ever false, and 32% were only ever true. This larger figure of 32% is in line with expectation, since it represents conditions which are true and iterations over non-empty collec-

UML Model	Percentage Coverage		
	ATL Rule	Instruction	Branch
1	20.0%	35.0%	26.2%
2	20.0%	35.0%	26.2%
3	30.0%	41.4%	29.1%
4	20.0%	35.3%	26.7%
5	30.0%	38.6%	27.2%
6	20.0%	35.0%	26.2%
7	10.0%	32.4%	25.2%
8	20.0%	35.5%	25.7%
9	20.0%	37.6%	25.7%
10	40.0%	57.1%	35.9%
11	50.0%	61.9%	38.8%
12	10.0%	28.5%	20.4%
13	20.0%	33.5%	23.3%
14	30.0%	36.6%	24.3%
15	10.0%	29.3%	21.8%
16	10.0%	28.5%	20.4%
17	30.0%	40.8%	27.7%
18	20.0%	35.0%	26.0%
19	30.0%	34.4%	22.8%
20	20.0%	33.5%	23.3%
Cum. Total	*80.0%*	*76.0%*	*51.0%*

Table 5.2: A summary of the percentage coverage for each of the test cases in Table 5.1. *This table summarises the coverage for each of the 20 UML models in the* UMLClass-Diag2Measurement *transformation test suite.*

tions, which is almost certainly the "intended" function of the code in each case. Nonetheless, the data of Table 5.2 provided the basis for developing a test suite that ensures better coverage.

On inspecting the ATL code corresponding to gaps in rule, instruction and branch coverage we were able to create UML class diagrams to force these items of the transformation to be exercised. To create the UML class diagram we used a UML modelling tool that supports the export of UML models as instances of the UML2 Eclipse metamodel called Papyrus [CEA].

In this case it was necessary to create two further models, an empty UML class diagram (model 22) and a UML class diagram with the following elements (model 21):

- An enumeration

Branch Evaluation	No. of branch instructions		
	IF	ITERATE	Total
Never executed	16	9	25 (= 24%)
Only ever false	6	12	18 (= 18%)
Only ever true	15	18	33 (= 32%)
Both true and false	14	13	27 (= 26%)
Total branch instrs.	*51*	*52*	*103 (= 100%)*

Table 5.3: A breakdown of the overall cumulative branch coverage data for all 20 UML models. *This table splits the branch instructions into four categories based on the degree to which they were covered during the transformations.*

- An interface with attributes
- A class and an association class that inherit from another class
- A class and an association class that implement an interface
- An abstract operation with a return parameter
- A class that overrides an abstract operation
- Several operations, each of which satisfies each of the conditions outlined for identifying the operation type.

With the addition of these final test cases to the test suite of Table 5.1 all 22 models were put through the testing process again and all three coverage figures were brought to the maximum coverage possible. In creating these two final UML class diagrams to ensure maximum coverage we exposed a further five errors in our transformation. Four of these errors were found in the rule for computing the type of a method. As this process involved extracting a substring from the name of the operation we discovered that the indexing being used to extract the names of the operations were incorrect in some cases and thus incorrectly identifying the type of a method. The final error was as a result of operations of an interface being omitted from the inherited-member set of any classes that implement the interface which in turn resulted in these operations being excluded from the set of overridden operations of that class. The second criteria for checking the correctness of the criteria is to check that the metric values calculated for the models are correct.

Test Oracle Construction

After all the errors described above had been corrected, all 22 models were put through our transformation again. On this final attempt all target models conformed

Test Case	Number of Classes	Attribs	Methods	Params	User-Def	BuiltIn	Max. DIT
1	2	1	0	0	0	0	0
2	4	2	0	0	0	0	0
3	3	1	0	0	0	0	0
4	4	1	0	0	0	0	1
5	10	4	0	0	0	0	0
6	2	1	0	0	0	0	0
7	4	0	0	0	0	0	1
8	4	3	0	0	0	0	0
9	3	0	0	0	0	0	0
10	4	4	3	4	0	0	0
11	4	10	1	1	0	1	1
12	1	0	0	0	0	0	0
13	1	1	0	0	0	0	0
14	1	1	0	0	1	0	0
15	4	0	0	0	0	0	1
16	1	0	0	0	0	0	0
17	3	1	0	0	0	0	0
18	1	1	0	0	0	0	0
19	3	0	0	0	1	2	0
20	4	1	0	0	0	0	0
21	6	2	12	4	1	0	2
22	0	0	0	0	0	0	0

Table 5.4: Metric results for the test suite. *This table lists some basic (and easily verifiable) object-oriented metrics for each of the 20 UML models from Table 5.1 and test cases 21 and 22, the additional models created to increase the coverage to the maximum possible.*

to the well-formed rules of the measurement metamodel. The final validation step was to use the resulting 22 instances of the measurement metamodel as input to the automatically generated measurement tool.

While this tool calculates a comprehensive set of metrics, we display the values of seven of them in Table 5.4 to give an estimate of the size of the class diagrams in the test suite. These metrics are a count of the various different elements in the UML class diagram, as calculated using the measurement tool. The metrics include number of classes, number of attributes, number of methods, number of parameters, number of user-defined types and number of built-in types in the class diagram. We also report the maximum depth of inheritance (DIT) found in the class diagram.

Each of these metrics, along with the remaining metrics in the *CK*, *Coupling*

and *Cohesions* metric sets were calculated both manually and automatically for all 22 UML class diagrams. The metrics were calculated manually by visually inspecting each of the class diagrams and computed automatically using the automatically generated measurement tool. Both sets of metrics were then compared thus providing the final check for correctness of the transformation process. The metric results for all 22 UML class diagrams can be found on the disk accompanying this thesis.

5.6.2 Testing the *Java2Measurement* Transformation

This section details the results of using the testing strategy described in this chapter to the *Java2Measurement* transformation.

Test Suite

In order to test the transformation we first had to construct a test suite. To achieve this we took the set of 22 UML models used as a test suite for the *UMLClassDiag2Measurement* transformation and forward engineered these into Java using EclipseUML [Omo]. Again, there was no coverage data or analysis available for these models, however they were chosen as they they provided maximum posible coverage of the *UMLClassDiag2Measurement* transformation. Only 3 of the 22 models could be successfully imported into the EclipseUML tool. The remaining 18 models although conforming to the UML metamodel did not match the expected format of EclipseUML; for example it expects UML models to begin with a single named Model element. Fourteen models had to be modified to give their Model element a name and 4 of the models were modified to add an overall Model element. On modification of these models, 8 models still failed on import, this was discovered to be due to certain model elements not being suppoted. For example attributes with no types and multiplicities greater than 1. To overcome this, we gave the attributes a type and changed all multiplicities greater than 1 to 1. Now all 22 models could be successfully imported into EclipseUML which then automatically created the equivalent Java code. SpoonEMF has problems creating instances of the Java programs if they do not compile so we first compiled all 22 Java models and found that 4 models did not compile so these needed to be modified in order to compile. These 22 syntactically correct Java models were used as input to SpoonEMF and used to produce 22 instances of the Java metamodel which could then be used as a test suite for the *Java2Measurement* transformation. Using this test suite we

applied the testing strategy described earlier in this chapter and the results of this are outlined in the rest of this section.

Checking Metamodel Conformance

After the original set of test cases were put through the transformation, one model failed to produce a target model and the other 20 models produced violated the well-formedness rules of the measurement metamodel. In total 450 constraints were violated, but on further inspection it was found that only 8 of these constraints were unique. We traced this back to 3 errors, 1 in the model transformation and 2 in the Java metamodel.

One of the errors in the Java metamodel was found to be as a result of setting a method as abstract. As `abstract` is a keyword in Java it was not possible to use it as an enumeration literal and thus replaced it with the word `abst`. The other error was due to the `DeclaringPackage` reference of the `Package` element in the Java metamodel having its attribute `settable` set to false and therefore its value could not be set by ATL when performing the transformation. This was changed to be equal to true in the Java metamodel. The remaining error was found in the model transformation as a result of there being a distinct element for constructors in the Java metmodel and as a result they were not being included in the inherited methods of a class. This error was corrected in the Java transformation.

Coverage Analysis

The results of the coverage analysis are summarised in Table 5.5 on a per-model basis. This table has one row for each of the Java packages created from the UML models in Table 5.1. The data in each row represents the percentage coverage when the corresponding Java package was transformed into an instance of the measurement metamodel using the *Java2Measurement* transformation presented in this chapter. The final row of Table 5.5 shows the cumulative total coverage when all 22 Java packages were transformed using the *Java2Measurement* transformation, and thus represents the total coverage for all packages considered as a test suite.

In total the *Java2Measurement* transformation contained 11 ATL rules which, along with the helper operations, were implemented in 1876 virtual machine instructions, of which 86 were IF or ITERATE instructions, giving a total of 172 possible true/false branches.

Java Package	Percentage Coverage		
	ATL Rule	Instruction	Branch
1	45.5%	63.3%	44.8%
2	45.5%	63.3%	44.8%
3	45.5%	63.3%	44.8%
4	45.5%	71.6%	53.5%
5	45.5%	63.3%	44.8%
6	45.5%	63.3%	44.8%
7	18.2%	44.2%	37.2%
8	18.2%	35.3%	28.5%
9	18.2%	35.3%	28.5%
10	54.5%	68.6%	47.7%
11	63.6%	78.9%	57.6%
12	18.2%	35.8%	28.5%
13	18.2%	35.3%	28.5%
14	54.5%	65.8%	45.9%
15	18.2%	44.2%	37.2%
16	18.2%	35.3%	28.5%
17	45.5%	63.6%	45.3%
18	45.5%	63.6%	44.2%
19	18.2%	35.3%	28.5%
20	45.5%	63.3%	44.8%
21	54.5%	80.2%	60.5%
22	0.0%	13.6%	12.2%
Cum. Total	*81.8%*	*88.2%*	*69.8%*

Table 5.5: A summary of the percentage coverage for each of the Java packages generated from the UML models in Table 5.4 *This table summarises the coverage for each of the 22 Java packages in the Java2Measurement transformation test suite.*

Overall we can see from Table 5.5 that each individual Java package covers between 18% and 64% of the rules, between 14% and 79% of the instructions and between 12% and 38% of the branches. The range of values is again quite small and we also again observe a tendency for instruction and branch coverage to increase together.

The total coverage data, shown in the last row of Table 5.5, provides a measure of the effectiveness of the set of 22 Java packages considered as a test suite. As can be seen the cumulative coverage is quite high at 81.8% rule coverage and 88.2% instruction coverage, though somewhat lower for branch coverage at nearly 70%.

Further analysis of the low branch coverage data from Table 5.5 is given in Table

Branch Evaluation	No. of branch instructions		
	IF	ITERATE	Total
Never executed	5	1	6 (= 7%)
Only ever false	5	4	9 (= 11%)
Only ever true	10	21	31 (= 36%)
Both true and false	23	17	40 (= 47%)
Total branch instrs.	*43*	*43*	*86 (= 100%)*

Table 5.6: A breakdown of the overall cumulative branch coverage data for all 22 Java packages. *This table splits the branch instructions into four categories based on the degree to which they were covered during the transformations.*

5.6. From this table we can see that 7% of the missing coverage is due to branch instructions not being executed at all. A further 11% of branch instructions were only ever false, and 36% were only ever true. The data in Table 5.3 again provided us with information that was used as a basis to develop further test cases to bring the coverage values to the maximum coverage possible.

On inspecting the ATL code corresponding to gaps in rule, instruction and branch coverage we were able to create an extra Java package to force these items of the transformation to be exercised. We created a Java package with the following elements:

- An interface inheriting from another interface
- A method invocation
- A constructor invocation
- A field access
- A field that is an array of built in types
- A field that is an array with a class type
- An overridden method that is not abstract

In creating this final package we exposed a further error. This was found in the Java metamodel specification, the `Declaration` reference of the element `Reference` had its settable option set to false and thus the transformation failed when trying to instantiate the metamodel. The second criteria for checking the correctness of the criteria is to check that the metric values calculated for the models are correct.

Test Oracle Construction

After all the errors described above had been corrected, all 23 Java packages were put through our transformation again. On this final attempt all target models conformed to the well-formed rules of the measurement metamodel. The final validation step was to first manually calculate values for the metrics in the *CK*, *Coupling* and *Cohesions* metric sets for all 23 instances of the measurement metamodel and second to use these 23 measurement metamodel instances as input to the automatically generated measurement tool. Both sets of manually and automatically calculated metrics were compared thus providing the final check for correctness of the transformation. The metric results for all 23 Java packages can be found on the disk accompanying this thesis.

5.7 Discussion

In this chapter, we completed the measurement approach by introducing support for model transformations and defined the *UMLClassDiag2Measurement* transformation and the *Java2Measurement* transformation to illustrate how the approach can be applied to UML and Java. When defining these transformations, it was apparent that certain concepts of the source languages did not map directly to elements of the measurement metamodel. In these situations, decisions were made as to which elements of the measurement metamodel these concepts should map to. For example, in both transformations the decision was taken to map interfaces to classes in the measurement metamodel. This has important implications for some of the coupling metrics such as CBO and RFC' as alternative decisions would effect the metric values produced. Fundamentally, these type of decisions are necessary and will effect the metric values produced regardless of the measurement approach used. The advantage of the approach outlined in this thesis is that these decisions are transparent and made explicitly clear in the model transformations. Alternative decisions can be easily implemented using alternative model transformations and no modifications or re-implementation of the measurement tool is required.

A set of coverage criteria to measure the adequacy of a test suite used in testing model transformations have also been proposed in this chapter. We applied these coverage criteria to the *UMLClassDiag2Measurement* and *Java2Measurement* transformations in order to correct and refine the transformations while they were being

developed. We believe that these criteria are applicable to the development and testing of any model transformation.

We have developed a Coverage Analyser to measure these coverage criteria that analyses the log files produced duing the execution of a transformation on the ATL virtual machine. There are several other possible methods of measuring the coverage criteria such as instrumenting the ATL transformation source or compiled transformation (.asm) file. However this would be quite technically challenging to implement because this would require comprehensive knowledge of the ATL virtual machine and structure and format of the compiled ATL files.

We also proposed an iterative process for testing the correctness of the model transformation. This approach has been designed specifically to define and test any transformation where the target metamodel is the measurement metamodel. One of the criteria used to check the correctness of the transformation is that the metric values for the resulting target models are correct. At the moment, this checking is done by hand which is tedious and error-prone and does not scale well to very large number of test cases. An improvement to this would be to automate this process.

Furthermore, on inspection of the metric values for the models that satisy the transformation coverage criteria it can be noted that a large number of these metrics yield a value of 0 for many of the metrics. This is as a result of our preference for using a small number of test cases containing a small number of elements to make the process of manually checking the metrics easier. This raises the question of whether the transformation criteria alone are sufficient. We may also want models that give a range of metric values for particular metrics. Therefore, its possible that we also need to take into consideration the coverage of the *range of possible values* for the different metrics. For example, for coupling metrics, we might demand that there exist models that yield the minimum and maximum coupling values and maybe values in-between. Further, we might also like to show that, given different versions of the same metric, test cases exist that show these versions returning different values.

Finally, this chapter has focussed mainly on unit testing with the emphasis on the individual transformation rules and elements of the source metamodels. As already mentioned this required a small number of test cases containing a small number of elements. While this had the advantage of making it easy to manually inspect the models, verify the metrics and uncover errors, it is possible that the approach may have missed errors related to testing multiple units at once such as

excercising many transformation rules across a single model containing many or all types of elements from the source metamodel. Also, this approach does not test non-functional aspects of the system such as performance and scalability. Therefore this work would benefit from applying other types of testing techniques such as integration or system testing in order to further evaluate the reliability, scalability and performance of the model transformations and overall measurement approach.

5.8 Summary

Model transformations play a very important role in MDE. Since testing is essential at all stages of software development, it is important to ensure that model transformations can be validated. While there are several aspects to software testing, including the generation of test cases, test adequacy criteria and the definition of a test oracle, this chapter concentrates on coverage measures for model transformations.

In this chapter we have presented three transformation-based coverage measures that capture the dual nature of a model transformation: part input-recognition, like a grammar, and part generation, like program code. Thus we extend the notion of rule coverage from grammars to model transformations, and use instruction and branch coverage to evaluate the remaining elements. We have developed tool support to measure these criteria for the transformation language ATL. Finally, we describe how these criteria were used in the process of testing two model transformations, the *UMLClassDiag2Measurement* for transforming UML class diagrams to instances of the measurement metamodel and the *Java2Measurement* for transforming Java programs to instances of the measurement metamodel.

We identify the four principal contributions of this chapter as:

- The **definition of two model transformations**, *UMLClassDiag2Measurement* for converting UML class diagrams to instances of the measurement metamodel and *Java2Measurement* for converting Java prgrams to instances of the measurement metamodel, thus providing the ability to automatically measure a set of existing coupling and cohesion metrics from UML class diagrams and Java programs.
- A set of **model transformation based coverage criteria** for the ATL language that can be used to assess the adequacy of test cases used in testing model transformations.

- A **testing strategy** for testing the correctness of model transformations where the target metamodel is our measurement metamodel.
- An **illustration of the use** of the coverage criteria and testing strategy for model transformation testing using the *UMLClassDiag2Measurement* and *Java2Measurement* transformations.

Chapter 6

Concluding Remarks

In this thesis we have described an approach to software measurement based on MDE principles. We have described a number of techniques for testing within the domain of MDE and applied these techniques to our measurement approach. This final chapter identifies the contributions of this work and presents a discussion of future work.

6.1 Contributions

The work presented in this thesis makes several contributions to the fields of object-oriented software metrics and MDE. In this section we identify and summarise each of these contributions.

We began our work by addressing the problem of having multiple different definitions for software metrics by using the OCL as a way to define and implement object-oriented software metrics. We were the first authors to generalise this approach at the metamodel level and apply it to various different metamodels (e.g. Java and UML 2.0 metamodels). We developed a highly extensible approach for the definition and calculation of software metrics which allows multiple definitions of software metrics to be implemented and compared. We provided the first ever definitions of the CK metrics using both the UML 2.0 and Java metamodels [MP06d, MP06b].

We presented the first MOF-compliant metamodel for software measurement. This metamodel was based on a formalisation of the work of Briand *et al.* and is

specific to the measurement of coupling and cohesion metrics [MP08b, MP]. We also defined two model transformations for converting UML class diagrams and Java programs to instances of the measurement metamodel. These transformations provide an automated way to transform UML class diagrams and Java programs to instances of the measurement metamodel thus facilitating the automatic calculation of object-oriented metrics for those artifacts, provided the metrics have been defined using the measurement metamodel.

A further contribution of our work is a general approach for assessing and checking the correctness of a metamodel using an iterative approach of developing and validating the metamodel using the Alloy language and analyser. We also extended this approach to provide a mechanism for automatically generating test data for MOF metamodels. We illustrated the use of this approach in the development and testing of our MOF-compliant measurement metamodel and measurement tool generated from the metamodel [MP08b, MP]. To our knowledge, we are the first to propose and use the Alloy language and analyser for this purpose.

We defined two model transformations, specifically from the UML and Java metamodels to our measurement metamodel. One of the important tasks during the development of a model transformation is the validation and verification of the transformation. One approach to this is software testing which is used to determine if the transformation is correct i.e. whether the target models produced by the model transformation implementation satisfy the transformation specification [Lam07]. Determining appropriate methods for performing this type of transformation testing is currently an open question in the MDE community [BDTM$^+$06]. To address the problem in our case, we leveraged the fact that we were able to determine the results of the software metrics for the source models and used this information to assess if the target models produced by the transformations were correct. This novel approach to transformation validation allowed us to refine our transformations by identifying and correcting bugs and errors within our transformations. To measure the adequacy of the test cases used during this testing process, we proposed a set of novel criteria for measuring the coverage of the model transformation. Although this testing approach applies specifically to model transformations involving the measurement metamodel, we believe that these adequacy criteria are generally applicable to testing any model transformation.

6.2 Future Work

We can identify a number of additional issues related to the research presented in this thesis that we believe merit further investigation.

6.2.1 Improvements to the Measurement Approach

The tool support for the measurement approach relies on the Octopus tool to transform the OCL into a working Java implementation which then requires the user to instantiate the metamodel over which the metrics are defined. While this provides a flexible environment which can be used for any metamodel (provided it can be depicted as a class diagram) it is not consistent with a fully integrated MDE approach because this method seperates the definition and evaluation of the metrics from the process of transforming models to the measurement metamodel. For example, in Chapter 5 we used the EMF to express the measurement metamodel in Ecore format and transformed the other languages to the measurement metamodel using a transformation language that works for Ecore metamodels. The measurement metamodel over which the OCL metrics were expressed was supplied to Octopus in a format specific to the Octopus tool, thus we had to maintain the measurement metamodel in two different formats. Ideally, we would define and evaluate the OCL metric defintions directly over the Ecore measurement metamodel and Ecore models. To achieve this we would need to make use of the OCL tool support available for Ecore models.

Since the development of dMML there has been an initiative within the Eclipse community to develop OCL support for EMF/Ecore models as part of the Model Development Tools (MDT) project [Ecld]. This project provides a parser and interpreter as well as an OCL to Java compiler for OCL constraints and queries expressed over any EMF-based metamodel. It would be a useful endeavour to update the OCL component of dMML from Octopus to either the OCL to Java compiler or the OCL evaluator provided by this Eclipse project. This would provide the ability to express the OCL queries directly over the measurement metamodel represented as an Ecore metamodel thus creating a more integrated MDE approach.

It would still be neccessary however to test the measurement metamodel and the metric definitions to assess their correctness and the testing approach outlined in Chapter 4 could still be used for this purpose. However if an OCL evaluator was used, a new set of test adequacy criteria would have to be defined as the code

coverage criteria would no longer be appropriate. Since any MOF metamodel can be depicted as a UML class diagram, one option could be to use a set of existing coverage criteria for UML class diagrams as done by Fleurey *et al.* [FSB04]. They use the set of criteria proposed by Andrews *et al.* [AFGC03], including the Class Attribute (CA) and Association-end multiplicity (AEM) criteria. These metrics use the structure of the metamodel to define a set of coverage criteria.

However, one problem with the criteria of Andrews *et al.* is that they do not take into account the set of OCL constraints expressed over the MOF metamodel. A more appropriate approach would be to define coverage criteria directly at the OCL level. Given this notion of coverage criteria for the OCL metamodel constraints, such criteria could then be used to assess the adequacy of the test cases in place of the code coverage criteria. Then following the approach described in Chapter 4, the Alloy analyser could be used to generate many models as test cases, and filter these models on the coverage criteria.

Our studies have shown that several hundreds of thousands of small-scope Alloy models must be generated even to satisfy basic code coverage criteria. Ideally the criteria themselves would be used as a basis for the generation of the models. Thus the Alloy analyser, aware of the coverage criteria, would systematically generate test cases that cumulatively satisfied these criteria. It is not obvious, to say the least, how the Alloy analyser might be adapted to achieve this goal. Therefore, a further path that needs to be investigated is how to incorporate notions of coverage into the model generation process. Once defined, the OCL-based coverage criteria could be used as coverage measures for the metric definitions as well as for the metamodel constraints since the metric definitions are also expressed in OCL. Thus we might demand the formulation of a set of models that also cover the OCL metric definitions.

6.2.2 Further Applications of the Measurement Approach

We have observed previously that the majority of the existing metrics for UML models are primarily simple counting metrics (e.g. number of use-cases in a model) such as those described by Marchesi, Genero *et al.* and Kim and Boldyreff [Mar98, GMP02, KB02, MP07]. In addition, the proposal of new metrics for UML models has concentrated on only one or else a small number of the different diagrams and views available in an overall UML model.

There has been relatively little work on defining how to measure existing design metrics from all of these different diagrams and views [MP07]. Our measurement approach would be useful here as it could be used to specify how to measure the three sets of software metrics described in this thesis (CK, Coupling and Cohesion) from a collection of different UML diagrams. This would strongly promote the use of these metrics at the design level and would allow for earlier quality assessment of software [TC02]. To accomplish this we would extend our *UMLClass-Diag2Measurement* transformation, described in Chapter 5 to take into account all of the elements of the UML metamodel relevant to the diagrams under consideration.

As an example, we consider the CK metrics proposed to capture different aspects of an object-oriented design. Baroni *et al.* have formalised the CK metrics using the OCL and the UML 1.3 metamodel [BA03b]. We have also formalised the CK metrics using the OCL but have based our definitions first on the UML 2.0 metamodel and then in a language independent way using our measurement metamodel [MP06a, MP06d, MP08a]. These definitions specify how to obtain the CK metrics from class diagrams but do not take any of the other UML diagrams into consideration. As mentioned, our measurement approach could be used to define how to measure the CK metrics from all the diagrams of a UML model. Figure 6.1 reviews each of these metrics and briefly discusses which UML diagrams need to be examined in order to gain accurate measures of the metrics. In addition, it may be possible to obtain further information for the calculation of these metrics, e.g. method invocations and variable usages of methods and classes, by inspecting OCL constraints of the system. Interpreting such information requires further research.

Finally, it is important to note that this approach is applicable to many different types of models, languages and metrics. For example, it could be extended to other object-oriented languages such as C++ and Smalltalk. This would require further model transformations to be developed to transfer models in these languages to instances of the measurement metamodel. The approach could also be extended beyond the domain of object-oriented languages and metrics to other areas such as functional and domain-specific languages. This could be achieved by developing further measurement metamodels to capture the relevant terms and concepts under measurement in these areas and defining the model transformations from the source languages to the measurement metamodels.

However, there is a significant amount of practical work required within the

Weighted methods per class (WMC): This metric is concerned with the complexity of the methods within a given class. It is equal to the sum of the complexities of each method defined in a class. If we consider the complexity of each method to be unity then the WMC metric for a class is equal to the number of methods defined within that class. The WMC metric for a class can be obtained from the class diagrams of a UML model by identifying the class and counting the number of methods in that class.

Number of children (NOC): This is the number of immediate descendants of a given class, that is the number of classes which directly inherit from the class. Again, this metric can be measured for a class by taking the union of all the class diagrams in a UML model and examining the inheritance relationships of the class.

Coupling between object classes (CBO): Two classes are coupled to each other if a method of one class uses an instance variable or method of the other class. An estimate for this metric can be obtained from the class diagrams by counting all the classes to which the class has a relationship with. To obtain a more precise value, information from the behavioural diagrams can be taken into account in order to get information about the usage of instance variable and invocation of methods. For example, a sequence diagram gives direct information about the interactions between methods in different classes.

Depth of inheritance tree (DIT): This is a measure of the depth of a class in the inheritance tree. It is equal to the maximum length from the class to the root of the inheritance tree. This metric can be computed for a class by taking the union of all the class diagrams in a UML model and traversing the inheritance hierarchy of the class.

Response for a class (RFC): This is a measure of the number of methods that can potentially be invoked by an object of a given class. The number of methods for a class can be obtained from a class diagram, but the number of methods of other classes that are invoked by each of the methods in the class requires information about the behaviour of the class. This information can be derived by inspecting behavioural diagrams, such as sequence and communication to identify method invocations.

Lack of cohesion in methods (LCOM): Calculating the LCOM for a given class involves working out, for each possible pair of methods, whether the sets of instance variables accessed by each method have an non-empty intersection. In order, to compute a value for this metric, information on the usage of instance variables by the methods of a class is required. This information cannot be obtained from a class diagram. However, an upper bound for this metric can be computed using the number of methods in the class. Diagrams that contain information about variable usages, e.g. sequence diagrams can be used for computing this metric.

Figure 6.1: An overview of applying the CK metrics to UML models. *This figure reviews the diagrams in a UML model that can contribute to calculating the CK metrics [MP07].*

area of MDE if this expansion of the measurement approach is to be explored further. Using metamodels to describe programming languages is a relatively new concept and not all programming languages have well-defined, accepted metamodels. Also it is our experience that when a metamodel has been defined for a language, the availability of tools for converting models written in the language to instances of the language metamodel is limited. If the transition from traditional programming paradigms to the MDE paradigm is to be successful then it is vital that such metamodels, tools and model transformations between languages are developed and adopted by the communuty. For such artefacts to become the de facto standard they must be shown to be correct, fit for purpose and reliable. As mentioned already, one way to achieve this is through the use of software testing. Therefore, the approaches presented in this thesis for testing metamodels, metamodel-based software and model transformations can play an important role in developing these necessary artefacts and having them adopted by the community.

6.3 Summary

The work in this thesis has demonstrated the overall practicality and feasibility of applying an MDE-based approach to software measurement. Furthermore, we have provided a means to ensure that the approach is reliable and correctly calculates metrics by proposing techniques for testing the various parts of the approach. We believe that these techniques are also applicable in the wider context of MDE and not just the measurement approach which is important as the future of software development moves in the direction of MDE.

Appendix A

Measurement Metamodel Specification

The description of our metamodel follows a similar format to that used by the OMG for the specification of metamodels, in particular the UML metamodel specification [OMG07b]. Each concept in the metamodel is described in its own subsection which is broken down into several different parts corresponding to different aspects of the concept. In situations where an aspect does not apply to the concept it is ommitted entirely from the description of the concept. Each concept is represented as a metaclass in the metamodel and described using the following

- The subsection heading gives the formal name of the concept.
- The *Description* aspect gives a brief, informal description of the meaning of the concept. Any direct generalisations of the concept are also detailed here.
- The *Attributes* aspect specifies each of the attributes that are defined for that metaclass. Each attribute is specified by its formal name and type. This is followed by an informal description of the meaning of the attribute.
- The *Associations* aspect lists all of the association ends owned by that metaclass. Again, each one is specified by its formal name, its type, and multiplicity and followed by an informal description of it's meaning.
- The *Constraints* aspect lists all of the constraints that define the well-formedness rules of the concept. Each constraint consists of an informal description and a formal constraint expressed in OCL.

- The *Operations* aspect contains a list of all the operations that belong to the metaclass. These include utility and query operations. In all cases each operation is specified using OCL. The utility operations are specified using the def keyword and the query operations are specified using the $body$ keyword.

A.1 Attribute

Description

An attribute is an entity that describes a property of the class that it belongs to. It is a subclass of NamedElement.

Associations

- **type:Type[0..1]** : specifies the type of the attribute
- **referenced_by:Method[0..*]** : specifies the set of methods that reference the attribute
- **att_implementing_class:Class[1..1]** : specifies the class in which the attribute is implemented
- **att_declaring_class:Class[0..*]** : specifies the set of classes in which the attribute is declared

A.2 BuiltIn

Description

BuiltIn represents a basic type provided by the modelling language (e.g., integer, real, character, string). It is a subclass of Type.

A.3 Class

Description

A class describes a set of entities that share the same properties and behaviour. It is a subclass of Type.

Associations

- **child:Class[0..*]** : specifies the set of immediate descendents of this class
- **parent:Class[0..*]** : specifies the set of immediate ancestors of this class
- **friend_of:Class[0..*]** : specifies the set of classes that are granted access to the non-public elements of this class
- **grants_friendship:Class[0..*]** : specifies the set of classes to which this class has access to the non-public elements of
- **overridden_method:Method[0..*]** : specifies the set of methods that are overridden by this class
- **inherited_method:Method[0..*]** : specifies the set of methods that are inherited by this class
- **new_method:Method[0..*]** : specifies the set of methods that are created as new in this class
- **declared_method:Method[0..*]** : specifies the set of methods that are declared in this class
- **implemented_method:Method[0..*]** : specifies the set of methods that are implemented in this class
- **declared_att:Attribute[0..*]** : specifies the set of attributes that are declared in this class
- **implemented_att:Attribute[0..*]** : specifies the set of attributes that are implemented in this class

Constraints

- A class may not directly or indirectly inherit from itself

```
not self.Ancestors()->includes(self)
```

- A class may not directly be a friend of itself or grant friendship to itself

```
not self.friend_of->includes(self)
```

- The set of declared attributes of a class must equal all the implemented attributes of that classes ancestors

```
self.declared_att = self.Ancestors()
                    ->collect(i:Class|i.implemented_att)
                    ->asSet()
```

- The set of new methods, overridden methods and inherited methods of a class must be disjoint

```
self.new_method->intersection(self.overridden_method)
               ->intersection(inherited_method)->isEmpty()
```

- The set of implemented methods of a class must equal the set of non-abstract, overriding methods union the set of non-abstract new methods of the class

```
self.implemented_method =
self.new_method->union(self.overridden_method)
               ->select(m:Method | not m.isAbstract)
```

- The set of declared methods of a class must be equal to the set of new abstract methods union the set of inherited methods of the class

```
self.declared_method =
self.inherited_method->union(self.new_method
->select(m:Method|m.isAbstract))
```

- The sum of the inherited and overridden methods of a class must equal the number of methods of the parents

```
self.parent->notEmpty() implies
self.inherited_method->union(self.overridden_method)->size()
= self.parent->collect(i:Class|i.Methods())->asSet()->size()
```

- The set of inherited methods of a class must be a subset of the new and overriding methods of that classes ancestors

```
not self.inherited_method->isEmpty() implies
(self.Ancestors()->collect(i:Class|i.new_method)->asSet()
                 ->union(self.Ancestors()
                 ->collect(j:Class|j.overridden_method)
                 ->asSet()))
                 ->includesAll(self.inherited_method)
```

- If a class has no parentss then it cannot have any overridden methods

```
self.parent->isEmpty() implies self.overridden_method->isEmpty()
```

Operations

- The query A_d() returns the set of declared attributes of the Class.

```
Class::A_d():Set(Attribute)
body: self.declared_att
```

- The query A_i() returns the set of implemented attributes of the Class.

```
Class::A_i():Set(Attribute)
body: self.implemented_att
```

- The utility operation all_parents() returns the set of all direct and indirect ancestors of the Class.

```
def: all_parents(S:Set(Class)):Set(Class)
=self.parent->union((self.parent - S)
            ->collect(i:Class|i.all_parents(S->including(self))))
            ->asSet()
```

- The query Ancestors() returns the set of all direct and indirect ancestors of the Class.

```
Class::Ancestors():Set(Class)
body: self.all_parents(Set{})
```

- The query Attributes() returns the set of all attributes belonging to the Class.

```
Class::Attributes():Set(Attribute)
body: self.declared_att->union(self.implemented_att)
```

- The query Children() returns the set of all immediate descendents of the Class.

```
Class::Children():Set(Class)
body: self.child
```

- The utility operation all_children() returns the set of all direct and indirect descendents of the Class.

```
def: all_children(S:Set(Class)):Set(Class)
=self.child->union((self.child - S)
           ->collect(i:Class|i.all_children(S->including(self))))
           ->asSet()
```

- The query Descendents() returns the set ofall direct and indirect descendents of the Class.

```
Class::Descendents():Set(Class)
body: self.all_children(Set{})
```

- The query Friends() returns the set of direct friends of the Class.

```
Class::Friends():Set(Class)
body: self.friend_of
```

- The query FriendsInv() returns the set of inverse friends of the Class.

```
Class::FriendsInv():Set(Class)
body: self.grants_friendship
```

- The query M_d() returns the set of declared methods of the Class.

```
Class::M_d():Set(Method)
body: self.declared_method
```

- The query M_i() returns the set of implemented methods of the Class.

```
Class::M_i():Set(Method)
body: self.implemented_method
```

- The query M_inh() returns the set of inherited methods of the Class.

```
Class::M_inh():Set(Method)
body: self.inherited_method
```

- The query M_ovr() returns the set of overridden methods of the Class.

```
Class::M_ovr():Set(Method)
body: self.overridden_method
```

- The query M_new() returns the set of new methods of the Class.

```
Class::M_new():Set(Method)
body: self.new_method
```

- The query M_pub() returns the set of public methods of the Class.

```
Class::M_pub():Set(Method)
body: self.Methods()->select(m:Method|m.isPublic)
```

- The query M_npub() returns the set of nonpublic methods of the Class.

```
Class::M_npub():Set(Method)
body: self.Methods()->select(m:Method|not m.isPublic)
```

- The query Methods() returns the set of all methods that belong to the Class.

```
context Class::Methods():Set(Method)
body: self.declared_method->union(self.implemented_method)
```

- The query Parents() returns the set of direct ancestors of the Class.

```
Class::Parents():Set(Class)
body: self.parent
```

- The query uses(d) returns true if the Class uses attributes or methods belonging to the Class d.

```
Class::uses(d:Class):Boolean
body: self.implemented_method->collect(m:Method|m.PIM())
                             ->intersection(d.implemented_method)
                             ->notEmpty()
      or
      self.implemented_method->collect(m:Method|m.referenced_att)
                              ->intersection(d.implemented_att)
                              ->notEmpty()
```

A.4 FormalParameter

Description

FormalParameter represents an argument that is used to pass information in and out of a Method. It is a subclass of NamedElement.

Associations

- **type:Type[0..1]** : specifies the type of the parameter
- **param_of:Method[1..1]** : specifies the method that the parameter belongs to

A.5 Invocation

Description

An invocation represents a method call.

Attributes

- **type:InvocationType** - specifies the type of the invocation i.e. wether it is static or polymorphic.

Associations

- **caller:Method[1..1]** : specifies the method in which the method call appears
- **calee:Method[1..1]** : specifies the method that is being called
- **passes_pointer_to:Method[0..*]** : specifies the set of methods that are being passed as a pointer during the invocation

A.6 InvocationType

InvocationType is an enumeration type that specifies the literals for defining the type of a method invocation.

Description

InvocationType is an enumeration of the following literal values:

- **static** : Indicates that the method is invoked statically.
- **dynamic** : Indicates that the method is invoked polymorphically.

A.7 Method

Description

A callable function belonging to a class. It is a subclass of NamedElement.

Attributes

- **type:MethodType** - specifies the type of the method
- **isAbstract:Boolean** - indicates if the method is abstract or non-abstract
- **isPublic:Boolean** - indicates if the method is public or non-public

Associations

- **overriding_class:Class[0..1]** : specifies the class that overrides this method
- **inheriting_class:Class[0..*]** : specifies the set of classes that inherit this method
- **new_class:Class[0..1]** : specifies the class in which this method is first defined
- **method_declaring_class:Class[0..*]** : specifies the set of classes in which this method is declared
- **method_implementing_class:Class[0..1]** : specifies the class in which this method is implemented
- **param:Parameter[0..*]** : specifies the set of parameters of this method
- **referenced_att:Attribute[0..*]** : specifies the set of attributes referenced by this method
- **invokes:Invocation[0..*]** : specifies the set of invocations in which this method is the caller
- **invoked_by:Invocation[0..*]** : specifies the set of invocations in which this metod is the callee
- **passed_to:Invocation[0..*]** : specifies the set of invocations where this method is passed as a parameter

Constraints

- If a method references at least one attribute, then this method must be non-abstract

```
self.referenced_att->notEmpty() implies self.isAbstract = false
```

- If a method is abstract then it must not call any methods

```
self.isAbstract = true implies self.invokes->isEmpty()
```

- If a method has no implementing class then it must be abstract

```
self.method_implementing_class->isEmpty() implies
self.isAbstract = true
```

- If a method is abstract then it must have a new class

```
self.isAbstract = true implies not self.new_class->isEmpty()
```

- A method must have either an overriding class or a new class

```
not self.overriding_class->asSet()->isEmpty() implies
    self.new_class->asSet()->isEmpty()
and self.overriding_class->asSet()->isEmpty() implies
    not self.new_class->asSet()->isEmpty()
```

- If a method is a constructor then it must not be abstract

```
self.type = MethodType::constructor
implies self.isAbstract = false
```

Operations

- The utility operation stat_invoked() returns the methods statically invoked by the Method.

```
def: stat_invoked():Bag(Method)
= self.invokes
  ->select(i:Invocation|i.type=InvocationType::static)
  ->collect(j:Invocation | j.callee)
```

- The utility operation poly_invoked() returns the methods polymorphically invoked by the Method.

```
def: poly_invoked():Bag(Method)
= self.invokes
  ->select(i:Invocation|i.type=InvocationType::polymorphic)
  ->collect(j:Invocation | j.callee)
```

- The utility operation closureSIM() returns the methods directly and indirevtly statically invoked by the Method.

```
def: closureSIM(S:Set(Method)):Set(Method)
= self.SIM()->union((self.SIM()-S)
            ->collect(m:Method|m.closureSIM(S->including(self)))
            ->asSet())
```

- The utility operation closurePIM returns the methods directly and indeirectlypolymorphically invoked by the Method.

```
def: closurePIM(S:Set(Method)):Set(Method)
= self.PIM()->union((self.PIM()-S)
            ->collect(m:Method|m.closurePIM(S->including(self)))
            ->asSet())
```

- The query operation AR() returns the set of attributes refernced by the Method.

```
Method::AR():Set(Attribute)
body: self.referenced_att
```

- The query operation NPI(m) returns the number of polymorphic invocations of m by the Method.

```
Method::NPI(m:Method):Integer
body: self.poly_invoked()->count(m)
```

- The query operation NSI(m) returns the number of static invocations of m by the Method.

```
Method::NSI(m:Method):Integer
body: self.stat_invoked()->count(m)
```

- The query operation Par() returns the set of parameters of the Method.

```
Method::Par():Set(FormalParameter)
body: self.param
```

- The query operation PIM() returns the set of methods polymorphically invoked by the Method.

```
Method::PIM():Set(Method)
body: self.poly_invoked()->asSet()
```

- The query operation SIM() returns the set of methods statically invoked by the Method.

```
Method::SIM():Set(Method)
body: self.stat_invoked()->asSet()
```

- The query operation PIM_() returns the set of indirectly polymorphically invoked methods of the Method.

```
Method::PIM_():Set(Method)
body: self.closurePIM(Set{})
```

- The query operation SIM_() returns the set of indirectly statically invoked methods of the Method.

```
Method::SIM_():Set(Method)
body: self.closureSIM(Set{})
```

- The query operation PP(m) returns the number of invocations of the Method where a pointer to the Method m is passed to this Method.

```
Method::PP(m:Method):Integer
body: self.invoked_by
        ->select(i:Invocation|i.passes_pointer_to->includes(m))
        ->size()
```

A.8 MethodType

MethodType is an enumeration type that specifies the literals for defining the type of a Method.

Description

MethodType is an enumeration with the following literal values:

- **constructor:** Indicates that the method is a constructor i.e. a method that is called when an object is created

- **destructor:** Indicates that the method is a destructor i.e. a method that is called when an object is destroyed
- **accessor:** Indicates that the method provides access to the attributes of the class to which it belongs
- **mutator:** Indicates that the method is used to modify the attributes of the class to which it belongs
- **general:** Indicates that the method is a general method that does't fall into any of the above categories

A.9 NamedElement

Description

A NamedElement is an element that may have a name. NamedElement is an abstract metaclass.

Attributes

- **String:name** - specifies the name of the element

A.10 Type

Description

A type defines a set of values. Type is an abstract metaclass. It is a subclass of NamedElement.

A.11 UserDefined

Description

A user-defined type of global scope (e.g., records, enumerations). It is a subclass of Type.

Appendix B

Requirements for the Model Transformation Language

Based on the work of Mens *et al.* and our own reqiuirements we set out a list of criteria to help us choose a model transformation language to integrate into our measurement approach [MG06]. Table B.1 lists our set of requirements, each of the 17 transformation languages that were examined and how they evaluated according to the requirements. We began by evaluating each language according to the first criteria, then the next etc. and if we encountered that the language did not meet one of the requirements we stopped our evaluation and moved onto the next transformation language. Each requirement can be summarised as follows:

- **Open source:** The transformation tool should be freely available for download and use and provide access to the source code. We aim to integrate the transformation tool into our existing measurement approach. The existing tool support for the approach is open source and therefore any additional tools must also be open source and freely available to download.

- **Standardised:** The tool support for the language should be compliant with the relevant standards. In our case we require tool support that facilitates transformations defined over MOF and Ecore metamodels and supports XMI for the import and export of source and target models.

- **QVT based:** As we are choosing to work with the OMG's standard for metamodelling, MOF, it would be desirable to also use a model transformation

language that adheres to the OMG's standard for model transformation, the Queries, Views, Transformations (QVT) [OMG08b].

- **Creating/Reading/Updating/Deleting transformations (CRUD):** The transformation tool should provide a mechanism for defining new and updating existing transformations for a modelling languages. As Mens *et al.* note, this may be a trivial requirement for a transformation language, it is not so trivial for a transformation tool [MG06]. For example, consider a code generation tool that only supports transformations of UML to Java or refactoring tools which only come with a predefined set of refactoring transformations and no support for defining new refactorings [MG06].

- **Fully automated:** Our measurement approach aims to provide fully automated tool support for the calculation of object-oriented software metrics. Therefore, the chosen tool should fully automate the transformation once it has been defined and require no manual intervention to complete the transformation.

After narrowing down our choice to 7 transformation tools we evaluated the tools using two further criteria identified by Mens *et al.*, *Usability and usefulness* which means that a language or tool should serve a practical purpose and be intuitive and efficient to use and *Acceptability by user community*, that is how well it's been adopted by the community and how widespread it's use is [MG06]. As part of this evaluation we attempted to carry out some basic transformations with each of the 7 tools. Based on this we made the pragmatic decision to integrate the ATL language into our measurement approach as it was one of the few tools that could successfullly execute our basic transformations. Our decision was also based on the fact that ATL has evidence of a large user community and a rich selection of sample ATL transformations to avail of which are stored as a collection of transformations referred to as a transformation zoo [ATL, Ecla].

Transformation Language	Requirement: Open source	Standard -ised	CRUD	QVT based	Fully automated	Eclipse Version	URL
AGG	✓	✗	■	■	■	■	http://tfs.cs.tu-berlin.de/agg/
AtoM3	✓	✗	■	■	■	■	http://atom3.cs.mcgill.ca/index_html
ATL	✓	✓	✓	✓	✓	3.3	http://www.eclipse.org/m2m/atl/
Borland Together	✗	■	■	■	■	■	http://www.borland.com/us/products/together/
EclipseM2M	✓	✓	✓	✓	✓	3.5	http://www.eclipse.org/m2m/
Fujaba	✓	✗	■	■	■	■	http://www.fujaba.de/
JMI	✓	✗	■	■	■	■	http://java.sun.com/products/jmi/index.jsp
Kermeta	✓	✓	✓	✓	✓	3.3	http://www.kermeta.org/
mediniQVT	✓	✓	✓	✓	✓	3.3	http://projects.ikv.de/qvt
ModelMorf	✗	■	■	■	■	■	http://www.tcs-trddc.com/ModelMorf/
MOLA	✓	✓	✓	✓	✓	3.3	http://mola.mii.lu.lv/
MTF	✓	✓	✓	✓	✗	■	http://www.alphaworks.ibm.com/tech/mtf
OptimalJ	✗	■	■	■	■	■	http://www.compuware.com/
SiTRA	✓	✗	■	■	■	■	http://www.cs.bham.ac.uk/~bxb/SiTra.html
smartQVT	✓	✓	✓	✓	✓	3.3	http://smartqvt.elibel.tm.fr/
Tefkat	✓	✓	✓	✓	✓	3.2	http://tefkat.sourceforge.net/
VIATRA2	✓	✗	■	■	■	■	http://www.eclipse.org/gmt/VIATRA2/

Table B.1: List of model transformation languages. *This table summarises all of the model transformation languages and tools that were considered for inclusion in the measurement approach and how they evaluate according to our requirements.*

Bibliography

[ABE+06] D.H. Akehurst, B. Bordbar, M.J. Evans, W.G.J. Howells, and K.D. McDonald-Maier. SiTra: Simple Transformations in Java. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 351–264, Genova, Italy, October 2006.

[ABF04] E. Arisholm, L.C. Briand, and A. Føyen. Dynamic coupling measures for object-oriented software. *IEEE Transactions on Software Engineering*, 30(8):491–506, 2004.

[ABGR07] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: A challenging model transformation. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, volume 4735 of *Lecture Notes in Computer Science*, pages 436–450, Nashville, TN, USA, October 2007. Springer.

[Abr01] F.B. Abreu. Using OCL to formalize object oriented metrics definitions. Technical Report ES007/2001, INESC Software Engineering Group, Portugal, May 2001.

[AFGC03] A. Andrews, R. France, S. Ghosh, and G. Craig. Test adequacy criteria for UML design models. *Software Testing, Verification and Reliability*, 13(2):95–127, 2003.

[AGaE95] F.B. Abreu, M. Goulão, and R. Esteves. Toward the design quality evaluation of object-oriented software systems. In *Proceedings of the International Conference on Software Quality*, pages 44–57, Austin, Texas, USA, October 1995.

[AL97] J.R. Abounader and D.A. Lamb. Data Model for Object-Oriented Design Metrics. Technical report, Department of Computing and Information Science, Queen's University, Kingston, ON, Canada, October 1997.

[ATL] ATLAS Team. ATLAS Transformation Language (ATL) Official Website. Available from `http://www.eclipse.org/m2m/atl/`. Last accessed March 7, 2011.

[BA02] A.L. Baroni and F.B. Abreu. Formalizing object-oriented design metrics upon the UML meta-model. In *Brazilian Symposium on Software Engineering*, Gramado - RS, Brazil, October 2002.

[BA03a] A.L. Baroni and F.B. Abreu. A formal library for aiding metrics extraction. In *ECOOP Workshop on Object-Oriented Re-Engineering*, Darmstadt, Germany, July 2003.

[BA03b] A.L. Baroni and F.B. Abreu. An OCL-based formalization of the MOOSE metric suite. In *ECOOP Workshop on Quantative Approaches in Object-Oriented Software Engineering*, Darmstadt, Germany, July 2003.

[Bar02] A.L. Baroni. Formal definition of object-oriented design metrics. Master's thesis, Vrije Universiteit Brussel - Belgium, in collaboration with Ecole des Mines de Nantes - France and Universidade Nova de Lisboa - Portugal, August 2002.

[BBA02] A.L. Baroni, S. Braz, and F.B. Abreu. Using OCL to formalize object-oriented design metrics definitions. In *ECOOP Workshop on Quantative Approaches in Object-Oriented Software Engineering*, Malaga, Spain, June 2002.

[BBM96] V. Basili, L. Briand, and W.L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.

[BCE] BCEL. The Apache Jakarta project. Available from `http://jakarta.apache.org/bcel/`. Last accessed March 7, 2011.

[BDM97] L.C. Briand, P. Devanbu, and W. Melo. An investigation into coupling measures for C++. In *Proceedings of the International Conference on Software Engineering*, pages 412–421, Boston, USA, May 1997.

[BDTM+06] B. Baudry, T. Dinh-Trong, J.-M. Mottu, D. Simmonds, R. France, S. Ghosh, F. Fleurey, and Y. Le Traon. Model transformation testing challenges. In *ECMDA workshop on Integration of Model Driven Development and Model Driven Testing*, Bilbao, Spain, July 2006.

[BDW98] L.C. Briand, J.W. Daly, and J.K. Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1):65–117, 1998.

[BDW99] L.C. Briand, J.W. Daly, and J.K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.

[Bei90] B. Beizer. *Software Testing Techniques*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[Ber08] K. Berkenkötter. Reliable UML Models and Profiles. *Electronic Notes in Theoretical Computer Science*, 217:203–220, 2008.

[BFS+06] E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. Le Traon. Metamodel-based test generation for model transformations: An algorithm and a tool. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 85–94, Raleigh, NC, USA, November 2006.

[BGaA02] A.L. Baroni, M. Goulão, and F.B. Abreu. Avoiding the ambiguity of quantitative data extraction: An approach to improve the quality of metrics results. In *Work in Progress Session at the EUROMICRO Conference*, Dortmund, Germany, September 2002.

[Bin00] R.V. Binder. *Testing Object-Oriented Systems: Models, Patterns and Tools*. Addison-Wesley, 2000.

[BK95] J.M. Bieman and B.K. Kang. Cohesion and reuse in an object-oriented system. In *Proceedings of the ACM Symposium on Software Reusability*, pages 295–262, Seattle, Washington, USA, April 1995.

[BMB94] L.C. Briand, S. Morasca, and V. Basili. Defining and validating high-level design metrics. Technical Report CS-TR 3301, Department of Computer Science, University of Maryland, College Park, MD 20742, USA, 1994.

[BMB96] L.C. Briand, S. Morasca, and V.R. Basili. Property-based software engineering measurement. *IEEE Transactions on Software Engineering*, 22(1):68–86, 1996.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[BSM+04] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T.J. Grose. *Eclipse Modeling Framework*. Addison-Wesley, 2004.

[Bun77] M. Bunge. *Treatise on Basic Philosophy, Ontology I: The Furniture of the World*. Boston: Reidel, 1977.

[BWDP00] L.C. Briand, J.K. Wüst, J.W. Daly, and V. Porter. Exploring the relationship between design measures and software quality in object-oriented systems. *The Journal of Systems and Software*, 51(3):245–273, 2000.

[CDC98] S.R. Chidamber, D.P. Darcy, and C.F.Kemerer. Managerial use of metrics for object-oriented software: An exploratory analysis. *IEEE Transactions on Software Engineering*, 24(8):629–639, 1998.

[CEA] CEA. Papyrus for UML. Available from `http://www.papyrusuml.org`. Last accessed March 7, 2011.

[CH06] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[CK91] S.R. Chidamber and C.F. Kemerer. Towards a metrics suite for object oriented design. In *Proceedings of the International Conference*

on Object-Oriented Programming, Systems, Languages, and Applications, pages 197–211, Phoenix, Arizona, United States, October 1991.

[CK94] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[CY91] P. Coad and E. Yourdon. *Object-oriented design*. Yourdon Press Englewood Cliffs, NJ, 1991.

[DDN00] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. *ACM SIGPLAN Notices*, 35(10):166–177, 2000.

[Den03] G. Dennis. TSAFE: Building a trusted computing base for air traffic control software. Master's thesis, MIT, Cambridge, Massachusetts, USA, January 2003.

[Dol] M. Doliner. Cobertura. Available from `http://cobertura.sourceforge.net/`. version 1.9, Last accessed March 7, 2011.

[DSW03] J.S. Dong, J. Sun, and H. Wang. Checking and reasoning about semantic web through Alloy. In *Proceedings of Formal Methods Europe*, volume 2805 of *Lecture Notes in Computer Science*, pages 796–814, Pisa, Italy, September 2003.

[DTD01] S. Demeyer, S. Tichelaar, and S. Ducasse. FAMIX 2.1 - The FAMOOS Information Exchange Model. *Research report, University of Bern*, page 11, 2001.

[EBC05] M. English, J. Buckley, and T. Cahill. A friend in need is a friend indeed. In *International Symposium on Empirical Software Engineering*, pages 453–462, Noosa Heads, Australia, November 2005.

[Ecla] Eclipse Open Source Community. ATL transformation zoo. Available from http://www.eclipse.org/m2m/atl/atlTransformations/. Last accessed March 7, 2011.

[Eclb] Eclipse Open Source Community. Eclipse IDE. Available from `http://www.eclipse.org/`. Last accessed March 7, 2011.

[Eclc] Eclipse Open Source Community. Eclipse Modelling Framework Project. Available from http://www.eclipse.org/emf/. Last accessed March 7, 2011.

[Ecld] Eclipse Open Source Community. OCL Project. Available from http://www.eclipse.org/modeling/mdt/?project=ocl. Last accessed March 7, 2011.

[Ecle] Eclipse Open Source Community. UML2 Project. Available from http://www.eclipse.org/uml2/. Last accessed March 7, 2011.

[Eclf] Eclipse Open Source Community. UML2 Tools Project. Available from http://www.eclipse.org/modeling/mdt/?project=uml2tools. Last accessed March 7, 2011.

[EKTW06] K. Ehrig, J. Küster, G. Taentzer, and J. Winkelmann. Generating instance models from metamodels. In *Proceedings of the International Conference on Formal Methods for Open Object-Based Distributed Systems*, volume 4037 of *Lecture Notes in Computer Science*, pages 156–170, Bologna, Italy, June 2006.

[EWEBRF05] M.M. El-Wakil, A. El-Bastawisi, M.B. Riad, and A.A. Fahmy. A novel approach to formalize object-oriented design metrics. In *Evaluation and Assessment in Software Engineering*, Keele, UK, April 2005.

[Fav04] J.M. Favre. Towards a basic theory to model Model Driven Engineering. In *Workshop in Software Model Engineering (WiSME)*, October 2004.

[FP96] N. Fenton and S. Lawrence Pfleeger. *Software metrics: A rigorous and practical approach*. International Thompson Computer Press, 1996.

[FSB04] F. Fleurey, J. Steel, and B. Baudry. Validation in model-driven engineering: Testing model transformations. In *MoDeVa Workshop*, Rennes, France, November 2004.

[GaA04] M. Goulão and F.B. Abreu. Formalizing metrics for COTS. In *ICSE Workshop on Models and Processes for the Evaluation of COTS Components*, Edinburgh, Scotland, May 2004.

[GBR05] M. Gogolla, J. Bohling, and M. Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Journal of Systems and Software*, 4(4):386–398, 2005.

[GG75] J.B. Goodenough and S.L. Gerhart. Toward a theory of test data selection. In *Proceedings of the International Conference on Reliable Software*, pages 493–510, Los Angeles, California, 1975.

[GJP02] M. Genero, L. Jimnez, and M. Piattini. A controlled experiment for validating class diagram structural complexity metrics. In *International Conference on Object-Oriented Information Systems*, volume 2425 of *Lecture Notes in Computer Science*, pages 372–383, Montpellier, France, September 2002.

[GMP02] M. Genero, D. Miranda, and M. Piattini. Defining and validating metrics for UML statechart diagrams. In *ECOOP Workshop on Quantitative Approaches in Object-Oriented Engineering*, Malaga, Spain, June 2002.

[GPC00] M. Genero, M. Piattini, and C. Calero. Early measures for UML class diagrams. *L'Object*, 6(4):489–515, 2000.

[Hal77] M. Halstead. *Elements of software science*. Elsevier, North Holland, first edition, 1977.

[HGS93] M.J. Harrold, R. Gupta, and M.L. Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering Methodology*, 2(3):270–285, 1993.

[HM95] M. Hitz and B. Montazeri. Measuring coupling and cohesion in object-oriented systems. In *Proceedings of International Symposium on Applied Corporate Computing*, pages 25–27, Monterrey, Mexico, October 1995.

[HP08] M. Hennessy and J.F. Power. Analysing the effectiveness of rule-coverage as a reduction criterion for test suites of grammar-based software. *Empirical Software Engineering*, 13(4):343–368, 2008.

[HS96] B. Henderson-Sellers. *Software Metrics*. Prentice Hall, Hemel Hempstead, U.K., 1996.

[HW02] T.J. Harmer and F.G. Wilkie. An extensible metrics extraction environment for object-oriented programming languages. In *Proceedings of the International Workshop on Source Code Analysis and Manipulation*, pages 26–35, Montréal, Canada, October 2002.

[JA06] F. Jouault and F. Allilaire. The ATL virtual machine. Available from `http://www.eclipse.org/m2m/atl/doc/`, 2006.

[JAB+06] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. ATL: a QVT-like transformation language. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 719–720, Portland, OR, USA, October 2006.

[Jac06] D. Jackson. *Software abstractions: logic, language, and analysis*. The MIT Press, 2006.

[Jak] Jakarta. The Apache Jakarta project. Available from `http://jakarta.apache.org/`. Last accessed March 7, 2011.

[JK05] F. Jouault and I. Kurtev. Transforming models with ATL. In *MoDELs workshop on Model Transformations in Practice*, Montego Bay, Jamaica, October 2005.

[KAER06] J. M. Küster and M. Abd-El-Razik. Validation of model transformations - first experiences using a white box approach. In *Proceedings of Workshop on Model Design and Validation*, pages 62–77, Genova, Italy, October 2006.

[KB02] H. Kim and C. Boldyreff. Developing software metrics applicape to UML models. In *ECOOP Workshop on Quantitative Approaches in Object-Oriented Engineering*, Malaga, Spain, June 2002.

[Ken02] S. Kent. Model Driven Engineering. In *Proceedings of International Conference on Integrated Formal Methods*, volume 2335 of *Lecture Notes in Computer Science*, pages 286–298. Springer, 2002.

[KHL01] B.A. Kitchenham, R.T. Hughes, and S.G Linkman. Modeling software measurement data. *IEEE Transactions on Software Engineering*, 27(9):788–804, 2001.

[Kol02] R. Kollmann. *Design recovery techniques for object-oriented software systems*. PhD thesis, Universität Bremen, November 2002.

[KPF95] B. Kitchenham, S.L. Pfleeger, and N. Fenton. Towards a framework for software measurement validation. *IEEE Transactions on Software Engineering*, 21(12):929–944, 1995.

[LA05] L. Lavazza and A. Agostini. Automated measurement of UML models: an open toolset approach. *Journal of Object Technology*, 4(4):115–134, 2005.

[Läm01] R. Lämmel. Grammar testing. In *Fundamental Approaches to Software Engineering*, volume 2029 of *Lecture Notes in Computer Science*, pages 201–216, Genova, Italy, April 2001.

[Lam07] M. Lamari. Towards an automated test generation for the verification of model transformations. In *Proceedings of the ACM Symposium on Applied Computing*, pages 998–1005, Seoul, Korea, March 2007.

[LD02] M. Lanza and S. Ducasse. Beyond language independent object-oriented metrics: Model independent metrics. In *ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, Malaga, Spain, June 2002.

[LH93] W. Li and S. Henry. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23(2):111–122, 1993.

[LK94] M. Lorenz and J. Kidd. *Object-oriented software metrics*. Prentice Hall Object-Oriented Series, 1994.

[LLL08] R. Lincke, J. Lundberg, and W. Löwe. Comparing software metrics tools. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 131–142, Seattle, WA, USA, July 2008.

[LLWW95] Y.S. Lee, B.S. Liang, S.F. Wu, and F.J. Wang. Measuring the coupling and cohesion of an object-oriented program based on information flow. In *Proceedings of the International Conference on Software Quality*, page 8190, Maribor, Slovenia, November 1995.

[LS06] R. Lämmel and W. Schulte. Controllable combinatorial coverage in grammar-based testing. In *Proceedings of International Conference on Testing of Communicating Systems*, volume 3964 of *Lecture Notes in Computer Science*, pages 19–38, New York, USA, May 2006.

[LTP04] T.C. Lethbridge, S. Tichelaar, and E. Ploedereder. The Dagstuhl Middle Metamodel: A schema for reverse engineering. *Electronic Notes in Theoretical Computer Science*, 94:7–18, May 2004.

[LZG05] Y. Lin, J. Zhang, and J. Gray. A testing framework for model transformations. *Model-Driven Software Development - Research and Practice in Software Engineering*, pages 219–236, 2005.

[Mar98] M. Marchesi. OOA metrics for the Unified Modeling Language. In *Procedihgs of Euromicro Conference on Software Maintenance and Reengineering*, pages 67–73, Florence, Italy, March 1998.

[McC76] T. McCabe. A software complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308320, 1976.

[McC96] S. McConnell. Daily build and smoke test. *IEEE Software*, 13(4):143–144, 1996.

[MCF03] S.J. Mellor, A.N. Clark, and T. Futagami. Guest editors' introduction: Model-driven development. *IEEE Software*, 20(5):14–18, 2003.

[MFJ05] P.A. Muller, F. Fleurey, and J.-M. Jézéquel. Weaving executability into object-oriented meta-languages. In *Proceedings of the International Conference on Model Driven Engineering Languages and*

Systems, volume 3713 of *Lecture Notes in Computer Science*, pages 264–278, Montego Bay, Jamaica, October 2005. Springer.

[MG06] T. Mens and P. Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[MGB04] T. Massoni, R. Gheyi, and P. Borba. A UML class diagram analyzer. In *In Procedings of the International Workshop on Critical Systems Development with UML*, pages 143–153, Lisbon, Portugal, October 2004.

[MJCH08] M. Monperrus, J.-M. Jézéquel, J. Champeau, and B. Hoeltzener. A model-driven measurement approach. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, Toulouse, France, October 2008.

[ML02] T. Mens and M. Lanza. A graph-based metamodel for object-oriented software metrics. *Electronic Notes in Theoretical Computer Science*, 72(2), 2002.

[MMG05] C. Marinescu, R. Marinescu, and T. Girba. Towards a simplified implementation of object-oriented design metrics. In *Proceedings of IEEE METRICS*, page 11, Como, Italy, September 2005.

[MP] J.A. McQuillan and J.F. Power. Test-driven development of a metamodel for the measurement of object-oriented systems. Under review.

[MP05] J.A. McQuillan and J.F. Power. A survey of UML-based coverage criteria for software testing. Technical Report NUIM-CS-TR-2005-08, Department of Computer Science, National University of Ireland, Maynooth, September 2005.

[MP06a] J.A. McQuillan and J.F. Power. A definition of the Chidamber and Kemerer metrics suite for the Unified Modelling Language. Technical Report NUIM-CS-TR-2006-03, Department of Computer Science, NUI Maynooth, Co. Kildare, Ireland, October 2006.

[MP06b] J.A. McQuillan and J.F. Power. Experiences of using the Dagstuhl Middle Metamodel for defining software metrics. In *Proceedings of the Principles and Practices of Programming in Java*, pages 194–198, Manheim, Germany, August 30 - September 1 2006.

[MP06c] J.A. McQuillan and J.F. Power. Some observations on the application of software metrics to UML models. In *Model Size Metrics Workshop of IEEE/ACM International Conference on Model Driven Engineering Languages and Systems*, Genoa, Italy, October 2006.

[MP06d] J.A. McQuillan and J.F. Power. Towards re-usable metric definitions at the meta-level. In *PhD Workshop of the European Conference on Object-Oriented Programming*, Nantes, France, July 2006.

[MP07] J. A. McQuillan and J. F. Power. On the application of software metrics to UML models. In *Models in Software Engineering*, volume 4364 of *Lecture Notes in Computer Science*, pages 217–226. Springer, 2007.

[MP08a] J. A. McQuillan and J. F. Power. Specifying coupling and cohesion metrics using OCL and Alloy. Technical Report NUIM-CS-TR-2008-02, Department of Computer Science, NUI Maynooth, Co. Kildare, Ireland, May 2008.

[MP08b] J.A. McQuillan and J.F. Power. A metamodel for the measurement of object-oriented systems: An analysis using Alloy. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*, pages 288–297, Lillehammer, Norway, April 2008.

[Mye04] G.J. Myers. *The Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 2004.

[ND08] C. Noguera and L. Duchien. Annotation Framework Validation using Domain Models. In *Proceedings of Model-Driven Architecture-Foundations and Applications (ECMDA)*, pages 48–62, Berlin, Germany, June 2008.

[NW02] A. Naumenko and A. Wegmann. A metamodel for the Unified Modeling Language. In *Proceedings of the International Conference on The Unified Modeling Language*, volume 2460 of *Lecture Notes in Computer Science*, pages 2–17, Dresden, Germany, 2002.

[OB88] T.J. Ostrand and M.J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, 1988.

[Obj] Klasse Objecten. Octopus: OCL tool for precise UML specifications. Available from `http://www.klasse.nl/octopus/`. Last accessed March 7, 2011.

[OMG00] OMG. Unified Modeling Language (UML) Specification v1.3. Ref.: formal/00-03-01, March 2000.

[OMG01a] OMG. Model Driven Architecture. Doc. # omg/03-06-01, 2001. `http://www.omg.org/mda/`.

[OMG01b] OMG. Unified Modeling Language (UML) Specification v1.4. Ref.: formal/2001-09-67, September 2001.

[OMG02] OMG. MOF 2.0 Query/Views/Transformations RFP. Doc. # ad/2002-04-10, April 2002.

[OMG03] OMG. Unified Modeling Language (UML) Specification v1.5. Ref.: formal/2003-03-01, March 2003.

[OMG04] OMG. UML profile for Meta Object Facility (MOF) specification v1.0. Ref: formal/04-02-06, February 2004.

[OMG05a] OMG. Revised submission for mof 2.0 query/view/transformation rfp. Doc. # ad/2005-07-01, July 2005.

[OMG05b] OMG. Unified Modeling Language (UML) Superstructure Specification v2.0. Ref.: formal/05-07-04, August 2005.

[OMG06a] OMG. Meta Object Facility (MOF) Core Specification v2.0. Ref: formal/06-01-01, January 2006.

[OMG06b] OMG. Object Constraint Language (OCL) 2.0 Specification. Ref.: formal/2006-05-01, May 2006.

[OMG06c] OMG. Request For Proposal: Software Metrics Metamodel. OMG Document # admtf/2006-09-03, 2006.

[OMG07a] OMG. Unified Modeling Language (UML) Infrastructure Specification v2.1.2. Ref.: formal/2007-11-04, November 2007.

[OMG07b] OMG. Unified Modeling Language (UML) Superstructure Specification v2.1.1. Ref.: formal/07-02-05, Feburary 2007.

[OMG07c] OMG. Unified Modeling Language (UML) Superstructure Specification v2.1.2. Ref.: formal/2007-11-02, November 2007.

[OMG07d] OMG. XML Metadata Interchange v2.1.1. Ref.: formal/07-12-01, December 2007.

[OMG08a] OMG. Architecture-Driven Modernization (ADM): Software Metrics Meta-Model (SMM) Submission. OMG Document # admtf/08-02-01, 2008.

[OMG08b] OMG. Meta object facility (mof) 2.0 query/view/transformation specification. Doc. # formal/2008-04-03, April 2008.

[Omo] Omondo. EclipseUML. Available from `http://www.eclipsedownload.com/`. Last accessed March 7, 2011.

[Pla] Planet-MDE. Planet-MDE. Available from `http://planet-mde.org/`. Last accessed March 7, 2011.

[Pur72] P. Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12(3):366–375, 1972.

[Rei01] R. Reißing. Towards a model for object-oriented design measurement. In *ECOOP Workshop on Quantative Approaches in Object-Oriented Software Engineering*, Budapest, Hungary, June 2001.

[Sch06] D.C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2):25–31, February 2006.

[SDM06] SDMetrics. The software design metrics tool for the UML. Available from `http://www.sdmetrics.com/`, 2006. Last accessed March 7, 2011.

[Sei03] E. Seidewitz. What models mean. *IEEE Software*, 20(5):26–32, 2003.

[SK03] S. Sendall and W. Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE SOFTWARE*, 20(5):42–45, 2003.

[SMC74] W.P. Stevens, G.J. Myers, and L.L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[SSSV04] M. Scotto, A. Sillitt, G. Succi, and T. Vernazza. A relational approach to software metrics. In *Proceedings of the Symposium on Applied Computing*, pages 1536–1540, Nicosia, Cyprus, March 2004.

[TC02] M.-H. Tang and M.-H. Chen. Measuring OO design metrics from UML. In *Proceedings of the International Conference on The Unified Modeling Language*, volume 2460 of *Lecture Notes in Computer Science*, pages 368–382, Dresden, Germany, September 30 - October 4 2002.

[W3Ca] W3C. Extensible markup language (XML). Available from `http://www.w3.org/XML/`. Last accessed March 7, 2011.

[W3Cb] W3C. Resource description framework (RDF). Available from `http://www.w3.org/RDF/`. Last accessed March 7, 2011.

[Wey84] EJ Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1984.

[WH02] F.G. Wilkie and T.J. Harmer. Tool support for measuring complexity in heterogeneous object-oriented software. In *Proceedings of the International Conference on Software Maintenance*, pages 152–161, Montréal, Canada, October 2002.

[WK03] J. Warmer and A. Kleppe. *The Object Constraint Language: Getting your models ready for MDA*. Addison-Wesley, 2003.

[WKB03] J. Warmer, A. Kleppe, and W. Bast. *MDA Explained: The Model Driven Architecture - practice and promise*. Addison-Wesley, 2003.

[WKC08] J. Wang, S.K. Kim, and D. Carrington. Automatic generation of test models for model transformations. In *Proceedings of Australian Software Engineering Conference*, pages 432–440, Perth, Australia, March 2008.

[WTH] E.Y. Shung Wong, O. Tayeb, and M Herrmann. A Guide to Alloy. Available from http://www.doc.ic.ac.uk/project/examples/2007-/271j/suprema_on_alloy/Web/. Last accessed March 7, 2011.

[YC79] E. Yourdon and L.L. Constantine. *Structured Design*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1979.

[YWG05] T. Yi, F. Wu, and C. Gan. A comparison of metrics for UML class diagrams. *ACM SIGSOFT Software Engineering Notes*, 29(5):1–6, 2005.

[ZD06] A. Zito and J. Dingel. Modeling UML2 package merge with Alloy. In *Alloy Workshop of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, Portland, Oregon, US, November 2006.

[ZHM97] H. Zhu, P.A.V. Hall, and J.H.R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.