Abstract

Transformations play a central role in Model Driven Engineering. Similar to the development of other types of software, a transformation’s specification and implementation does not necessarily remain static over the course of a project’s lifetime; the transformation may develop incrementally and evolve. The goal of this thesis is to propose metrics that can be used to characterize the evolution of model transformations. To perform an initial demonstration of the metrics, this thesis considers an incrementally defined model transformation task. The transformation is implemented using two model transformation languages, a textual language and a graphical language, and metrics are extracted from the historical artifacts.

The thesis defines a set of change metrics based on an abstract syntax difference model. Language feature metrics are also defined for both transformation languages. A process for extracting model-based change metrics and language metrics from the abstract syntax of the transformation languages is introduced. The applicability of the metrics in characterizing changes is demonstrated using exploratory clustering analysis on a transformation task. We show how, for this transformation task using both languages, metrics derived from the difference model result in clusters that reflect characteristics of individual changes, in contrast to clusters obtained with language metrics.
Acknowledgments

I would first like to extend the utmost gratitude and thanks to my supervisor, Dr. Juergen Dingel for his great mentorship throughout the entire duration of my graduate program. I would also like to thank the members of my defense committee, Dr. James Cordy, Dr. Thomas Dean, and Dr. Robert Tennent, whose comments and suggestions were tremendously helpful in focusing the message of my research. Thank you to my colleagues in the Software Technology Lab, from whom I’ve learned a great deal and with whom I’ve shared many laughs. Many thanks go to my family and friends for being there for all the ups and downs of my research experiences. Finally, I would like to thank Julia, for being my guiding light and for helping me find the motivation to remain on track.
# Contents

Abstract i

Acknowledgments ii

Contents iii

List of Tables vi

List of Figures viii

Chapter 1: Introduction 1

1.1 Models in software development ............................................. 2
  1.1.1 Domain-Specific Languages ........................................... 5
  1.1.2 The role of model transformations ................................... 5
  1.1.3 M2M vs M2T .......................................................... 6
1.2 Problem ................................................................. 7
1.3 Objective ............................................................... 8
1.4 Contributions ............................................................ 9
1.5 Organization of Thesis .................................................... 11

Chapter 2: Background 12

2.1 Data Mining Framework .................................................. 13
  2.1.1 Business Understanding .............................................. 14
  2.1.2 Data Understanding ................................................... 16
  2.1.3 Data Preparation ...................................................... 16
  2.1.4 Modelling ............................................................. 17
  2.1.5 Evaluation ............................................................. 18
  2.1.6 Deployment ............................................................ 18
2.2 Mining Software Repositories ....................................... 19
2.3 Measuring Modelling Languages ..................................... 20
2.4 Model Transformation Languages .................................... 21
  2.4.1 ATLAS Transformation Language ................................. 23
List of Tables

3.1 Change steps for the completion of Milestone 1 . . . . . . . . . . . . 56
3.2 Change steps from Milestone 1 to Milestone 2 . . . . . . . . . . . . . 59
3.3 Change steps from Milestone 2 to Milestone 3 . . . . . . . . . . . . . 61
3.4 Change steps from Milestone 3 to Milestone 4 . . . . . . . . . . . . . 63
3.5 Change steps from Milestone 4 to Milestone 5 . . . . . . . . . . . . . 64

4.1 Static metrics for RSA and ATL artifacts. . . . . . . . . . . . . . . . 76
4.2 Basic change operations and where they apply. . . . . . . . . . . . . 80
4.3 Descriptive change metrics that characterize individual changes. . . 81
4.4 Artifacts collected for the demonstrated transformation task. . . . . 82

5.1 Metrics abbreviations . . . . . . . . . . . . . . . . . . . . . . . . . . . 89
5.2 Pearson correlation coefficients between ATL change metrics. . . . . 90
5.3 Pearson correlation coefficients between RSA change metrics. . . . . 90
5.4 Pearson correlation coefficients between ATL language metrics. . . . 91
5.5 Pearson correlation coefficients between RSA language metrics. . . . 91
5.6 Cluster membership for ATL change metrics dendrogram (h=4.5) . . 95
5.7 Cluster membership for ATL language metrics dendrogram (h=7), in-
                     c ... 95
5.8 Cluster membership for RSA change metrics dendrogram (h=6.1) . . 96
5.9 Cluster membership for RSA language metrics dendrogram (h=3.0), including corresponding rule count median and median absolute deviation.
# List of Figures

1.1 A technological perspective of the OMG MDA framework. . . . . . . . . 3
1.2 A high-level view of the Eclipse Modelling Project’s underlying tech-
nologies. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

2.1 CRISP-DM data mining methodology. . . . . . . . . . . . . . . . . . 15
2.2 The ATL source code editor, shown with the accompanying Outline
view (right) and Problems view (bottom). . . . . . . . . . . . . . . . . 29
2.3 Eclipse Debug perspective showing a paused execution of an ATL
transformation. The current execution position is shown in the code
editor (bottom), and the context of the transformation at that execu-
tion position is shown in the Variable view (top-right). . . . . . . . . 30
2.4 An example of a transformation mapping definition in the RSA map-
ning tool. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
2.5 The main panel of a transformation configuration used to invoke an
RSA mapping. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
2.6 The "Source and Target" panel of a mapping transformation configura-
tion. This panel allows the user to select the input and output models
for which the transformation will execute. . . . . . . . . . . . . . . . . 39
3.1 The full process of the implemented transformation. Bolded parts indicate the focus of this thesis.

3.2 A simple example of a UML-RT state machine.

3.3 A view of the state structure classes in the simplified UML-RT metamodel.

3.4 A view of the classes related to transitions in the simplified UML-RT metamodel, as used in the transformation task.

3.5 Basic kiltera process constructs (a), and basic kiltera program structure constructs (b).

3.6 kiltera processes for data flow (a) and control flow (b).

3.7 The incremental development of UML-RT language features defined in the transformation’s mapping specification, by milestone.

3.8 UML-RT state machine example for Milestone 1: states n2 and n3 are nested within state n1, the initial transition of state n0 targets state n1, and the initial transition of state n0 targets state n2.

3.9 An excerpt of the specification containing the mapping definition that corresponds to Milestone 1.

3.10 UML-RT state machine example for Milestone 2: t1 is a ‘basic’ transition.

3.11 An excerpt of the specification that shows the mapping definition of Milestone 2.

3.12 UML-RT state machine example for Milestone 3: the execution of transitions t1 and t3 continue along different paths inside state n2 because they target different entry points.
3.13 UML-RT state machine example for Milestone 4: transitions t1 and t2 associated with different exit points from state n1 ........................................ 62
3.14 UML-RT state machine example for Milestone 5: t1 is a group transition 63
4.1 The data gathering process. Versions of transformation implementation artifacts are converted to an abstract representation. Difference models are calculated from adjacent versions. ........................................ 65
4.2 The process used by EMF Compare to generate its output models. . . 70
4.3 An example model modification: adding a Helper to the Module root. Two separate MatchEngine behaviours are shown. Matches are denoted by dashed lines. Detected changes are denoted by labels beside the elements. ................................................................. 70
4.4 Example model modification: changing the type of an element from MatchedRule to LazyMatchedRule. Matches are denoted by dashed lines. Detected changes are denoted by labels beside the elements. . . 73
4.5 The workflow for exporting repository artifacts .......................... 74
4.6 Change metrics output selection dialog. ................................. 75
4.7 Overall commit size for RSA Mapping data and ATL data ........... 83
5.1 Histogram and estimated density plot for depth_avg at commit granularity and at change step granularity ........................................ 88
5.2 Dendrograms for (a) ATL change metrics and (b) ATL language metrics 94
5.3 Dendrograms for (a) RSA change metrics and (b) RSA language metrics 97
5.4 (a) Cross-validation results for ATL change metrics and (b) corresponding optimal decision tree. .................................................. 99
Chapter 1

Introduction

Model transformations play a central role [49] in Model Based Software Engineering (MBSE), an approach for software engineering and software development. Many transformation languages already exist, and evaluation criteria and comparisons have been proposed to help developers find the language most suitable for a transformation task. However, these criteria and comparisons tend to consider the transformation implementation as fixed. In reality, a transformation’s artifacts can incrementally change and evolve like other software development artifacts.

This thesis explores the measurement of change in model transformation artifacts, as a foundation for the evaluation of model transformations and model transformation languages. The research is motivated by a real-world transformation task to define the formal semantics for an industrially used modelling language. It was found that the transformation developed incrementally, adding support for more language features over time. The overarching goal that we have only just begun to address with this research is the use of historical software development data to help manage the model transformation development process. For example, it is possible that trends and patterns detected in historical data could aid in predicting the effort of planned
transformation mappings or flagging defect “hot-spots”.

1.1 Models in software development

Before introducing model transformations in more detail, it is necessary to explain how MBSE works and to explain the purpose of models. MBSE is based on the tenet that models are the central artifacts of the development process. These models usually provide an abstract representation of the software being developed, compared to artifacts such as implementation code and compiled target executables. Models are typically said to conform to a “metamodel”, which is a description of the structure and semantics of model instances. Modern implementations of MBSE have in large part evolved from Computer-Aided Software Engineering (CASE) tools, which were conceived in the 1980s as a way to apply more automation to the specification and implementation of software systems. Selic [51] covers some of the key areas where modern varieties of MBSE have tried to improve on CASE:

- interoperability and standardized model representation,
- more rigorous approaches to modelling language design, and
- more rigorous approaches to the transformation and analysis of models.

Even though these issues are acknowledged and addressed to some extent, it appears that they remain open problems going forward. Selic also mentions that usability and user interaction still seems to be a secondary concern for teams that implement MBSE tools. This may partly be attributed to accidental complexity, introduced because of poor user interaction design. However, it is also possible that these usability issues can be caused by a more fundamental lack of understanding of
the theory of software modelling. Thus, even though modern MBSE methodologies might be considered a better attempt at the objectives that CASE originally tried to meet, there is still room for improvement.

Many frameworks have been developed for software modelling. In commercial and industrial software engineering, two major frameworks used are Model Driven Architecture (MDA), developed by the Object Management Group (OMG), and the Eclipse Modelling Project, developed by the Eclipse foundation. The MDA framework revolves around a standard methodology whereby Platform-Independent Models (PIMs) are gradually refined into Platform-Specific Models (PSMs), and finally into implementation code. The technology underlying MDA’s model representations are based on the Unified Modelling Language (UML) and the XML Metadata Interchange (XMI) format. UML is the “working” language of MDA, of which the Metadata Object Format (MOF) is a subset that is used for metamodelling. XMI is a storage representation of model data.

<table>
<thead>
<tr>
<th>Tools</th>
<th>Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>UML + Profiles</td>
<td>MOF</td>
</tr>
<tr>
<td></td>
<td>XMI</td>
</tr>
</tbody>
</table>

Figure 1.1: A technological perspective of the OMG MDA framework.

The Eclipse Modelling project is a set of technologies and tools centered around a modelling framework called Eclipse Modelling Framework (EMF) that has strong similarities to the corresponding part of the OMG technology stack. At the bottom of the stack, EMF also utilizes XMI as a model storage and interchange format. On
1.1. MODELS IN SOFTWARE DEVELOPMENT

Top of XMI, the ecore meta-metamodel provides a similar facility to MOF. Finally, tools for modelling and metamodelling are built on top of EMF and integrated into the Eclipse application platform.

<table>
<thead>
<tr>
<th>Tools</th>
<th>Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMF-based Metamodels</td>
<td>Ecore/EMF</td>
</tr>
<tr>
<td>XMI</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1.2: A high-level view of the Eclipse Modelling Project’s underlying technologies.

There are also many approaches to MBSE with origins in the academic realm. Two of the better known ones are “A Tool for Multi-formalism Meta-Modelling” (AToM³) and Algebraic Specification Formalism-Syntax Definition Formalism (ASF-SDF). AToM³ is a graphical metamodelling platform with support for graph-based model transformations [52]. ASF-SDF, on the other hand, is rooted in meta-programming and source rewriting concepts [14]. Though not strictly developed from within the MBSE realm, it is a text-based meta-programming language with a workbench called “Meta-Environment” that provides facilities to create and transform textual modelling languages.

Comparing these MBSE frameworks, we can see that there are some commonalities. All of the approaches have a concept of metamodelling to support the structured definition of model instances. All of the approaches mentioned here provide tools to take advantage of high-level models for automation and for abstraction.
1.1. MODELS IN SOFTWARE DEVELOPMENT

In our research, we concentrate on the EMF-based modelling platform, since it is the one that is used in the transformation task used to demonstrate this research. Further, there is an active ecosystem of technologies and tools (both academic and industrial) surrounding it.

1.1.1 Domain-Specific Languages

The concept of Domain-Specific Languages (DSLs) emerged from programming language design as a way to extend the syntax and semantics of more general languages. Recently, research interest in DSLs has increased due to its close relation to software (and domain) modelling. In a sense, “metamodelling” can be considered a form of DSL creation in an MBSE context. In terms of MBSE, a DSL provides the syntax and semantics of an abstract meta-model that represents a higher-level domain.

Typically, metamodelling tools and language workbenches provide mechanisms to separate the underlying abstract syntax and semantics of DSLs from the concrete syntax that is presented to users of the language. DSLs are also generally smaller than general-purpose languages in terms of scope and grammar. Combined, these two factors give designers of DSLs considerable flexibility in defining concrete syntax. Modelling languages might have graphical forms, textual forms, mixes of the two, or even multiple combinations of alternate syntaxes.

1.1.2 The role of model transformations

One crucial piece of the MBSE puzzle remains: some sort of machinery must be in place so that users of the modelling platform can operate on their models. It is this machinery that enables different abstractions and different representations of models.
1.1. MODELS IN SOFTWARE DEVELOPMENT

In other words, the role of model transformations is to exploit the abstraction potential of models. In MBSE, model transformations can be used to realize metamodel semantics and tooling, to facilitate interoperability between DSLs, and to perform analysis on models.

Model transformation languages provide software designers with a way to specify transformations. Model transformation languages can, in a sense, be considered domain-specific languages where the domain is model transformations. There are many applications where the transformations are mappings from a set of source models to a set of target models, so many transformation languages provide explicit mechanisms to define these mappings. Most transformation languages also incorporate some form of expression language so that transformation writers can navigate and reference elements in the source and target models. Sometimes the expression languages also provide general-purpose programming constructs, widening the applicability of the transformation language to a larger range of transformation tasks.

1.1.3 M2M vs M2T

Model transformation languages can be separated into two broad categories; Model-to-Model (M2M) transformation languages and Model-to-Text (M2T) transformation languages. As the name implies, M2M transformations allow operations to be performed on input source models, thereby outputting modified source models or new target models. Examples of use for M2M transformations can include model refinements such as those prescribed by the MDA approach and analysis techniques where properties of models are captured in abstractions. In contrast, the output target of
an M2T transformation is a text artifact. Examples of text artifacts might be: generated code, text-based human-readable reports, and serialization formats that enable interoperability with tools outside the MBSE framework (for example analysis tools that have not been designed to integrate with any modelling platforms).

In this thesis we will focus on M2M languages, simply because the demonstrated transformation task is a model-to-model transformation. However, the concepts that are introduced can still apply to M2T languages as long as a sufficient abstract representation of the language is available. Section 2.4 covers in more detail the classification of model transformation languages as it relates to the demonstrated transformation task.

1.2 Problem

In designing a modelling language, the focus tends to be on the language’s syntax and its semantics. There is a third aspect that is related to both: the “pragmatics” of the language. The pragmatics of the language describe how the language is used in the field. This can include both the relevant application domain as well as user interaction with the language’s tooling.

One of the major barriers for the adoption of model-based development paradigms in industry has historically been a lack of good tooling and difficulty in integrating tools efficiently [51]. It seems that investigation of the pragmatics of modelling languages is a useful exercise, in addition to making sure that the domain is accurately modelled by syntax and semantics. This holds especially for modelling languages because of their emphasis on facilitating human communication. Prospective users of these languages (typically domain experts) may not be familiar with some of the
1.3. OBJECTIVE

interfaces and concepts that developers and computer scientists are comfortable with, so it is important to account for all audiences that will use the language and its tools.

In long-running MBSE projects, especially involving DSLs, it may be useful to monitor the role of evolving language usage in the quality of software development artifacts. DSLs are more likely to be developed in-house, tailored to more specific technical and organizational requirements than general purpose languages. If these requirements change, either an adaptation of the language or an adaption of the usage of the language would be necessary.

The study of the usage of transformation languages may also be important because of the centrality of these languages in the MBSE process. Some researchers argue that the quality of transformation artifacts have an impact on the overall quality of models that are created and processed throughout the MBSE process [43].

1.3 Objective

The premise of this thesis is that it might be useful to characterize and analyze the way in which transformation implementations evolve in the context of their implementation languages. This study of language usage over time relates to the pragmatics of modelling languages as well as syntax and semantics. The contributions from this research and more extensive studies that may follow could help both users of transformation languages as well as transformation language designers.

The motivating long-term goal for this research, explained at the beginning of this chapter, is a continuous process to monitor historical development data, providing management assistance such as effort and defect prediction. The study of language
usage could also help to evaluate the application of transformation languages to dif-
ferent transformation tasks. Additionally, feedback from historical data can be used
by language designers, allowing them for example to tailor language features to com-
mon usage and to better understand the impact of new features. There is existing
work that defines language metrics for transformation languages which are based on
common Object-Oriented metrics, but the work so far only considers non-changing
transformation artifacts.

Some initial research questions can be formed from the motivating long-term goal:

1. Can direct measurements of change at the abstract syntax level be used to
characterise modelling language usage?

2. Can differences be observed in exploratory analysis between the application of
language artifact measurements and the application of change measurements?

Given the research questions, the objectives within the scope of this thesis are to:

• identify metrics of transformation language artifacts that may be related to the
  incremental development and evolution of said transformation artifacts,

• propose an approach for investigating modelling language usage over time,

• demonstrate the framework on a transformation task, and

• explore some change characteristics of the transformation task through simple
  preliminary analysis.

1.4 Contributions

In this thesis, the following contributions are presented:
1.4. CONTRIBUTIONS

- Preliminary work to develop a framework for analyzing and comparing model transformation languages and transformation artifacts, using change metrics to consider incremental development and evolution.
  
  - Definition of change metrics
  
  - Proposal for an approach to measure modelling language usage in a quantitative manner, borrowing concepts from the software engineering field called Mining Software Repositories (MSR)

- Application of the approach to an incrementally defined model transformation task
  
  - Method to define, at the specification level, the changes used to implement each iteration of the model transformation task
  
  - An account of the challenges associated with calculating metrics from graph-based change data
  
  - Simple exploratory analysis and evaluation of the model transformation using metrics

Though a simple exploratory analysis demonstrates how the metrics may be applied, the intent of the work presented in this thesis is not to provide a full empirical study analysing how transformations incrementally change in relation to quality properties. Instead, this thesis is a first step for more rigorous empirical exploration. The process that is introduced in this thesis to measure transformation languages can be used as a basis for studying evolution in model transformation languages, however the focus of the demonstrated transformation task is incremental development following the staged model [9], not strictly evolution in a post-deployment sense.
1.5 Organization of Thesis

We proceed by introducing an overall framework for the research in Chapter 2. Chapter 2 also provides the necessary context for the rest of the thesis. Chapter 3 describes a transformation task by examining the details of an incremental transformation specification. Chapter 4 describes a process for data collection, noting special procedures specific to the demonstrated transformation task and chosen transformation languages. Chapter 5 delves into some potential uses for the data and also describes how initial exploratory data was used to guide later iterations of analysis. Finally, Chapter 6 discusses the contributions of this thesis as well as possible future directions.
Chapter 2

Background

In this chapter we review the processes and concepts employed while conducting this research. Two general fields of software research are drawn from:

1. data mining, and

2. software modelling.

We begin with a high-level description of the data mining approach taken during the application of the process to an example transformation task. Then a treatment of the field of MSR is given, to provide more specific context for the data exploration and analysis techniques employed in Chapters 4 and 5. Along the modelling aspect, we expand upon the concepts introduced in Chapter 1. We cover some of the frameworks designed for model-based development approaches, and we also take a look at some of the efforts made to evaluate model transformation languages and modelling languages in general. Finally, we examine the two modelling languages that are used in the demonstration.
2.1 Data Mining Framework

A typical data mining approach might be based on the Cross Industry Standard Process for Data Mining (CRISP-DM) methodology [15], as shown in Figure 2.1. CRISP-DM is a frequently used methodology released by a consortium of industrial partners in 1999. The methodology is informed by industry practice, but it is shown to be quite similar to methodologies proposed in an academic context such as Knowledge Database Discovery (KDD) and “Sample, Explore, Modify, Model, Assess” (SEMMA) [6]. The methodologies differ mainly in scope; whereas KDD and SEMMA are data-centric, CRISP-DM also covers aspects of project management. Also, SEMMA is tailored to a specific analytics tool called SAS, whereas KDD and CRISP-DM are neutral with respect to tool vendors. This section explains the CRISP-DM methodology in the context of the research work.

The CRISP-DM approach is meant to be iterative and fluid, so the path along the phases is rarely linear. The arrows do not follow any particular order and the next iteration can begin anywhere within the process. For example, one might uncover unforeseen issues during data preparation, and would therefore need to revisit the data understanding or business understanding phases to update expectations and objectives. The modelling phase may require some extra data not extracted during data preparation, and therefore an phase of data preparation may be required after reviewing data understanding.

The work in this research focuses on the first phases of the methodology, i.e. “Business Understanding”, “Data Understanding”, and “Data Preparation”. Reports on exploratory observations can be considered part of the “Deployment” phase. The “Modelling” and “Evaluation” phases are not as extensively carried out in our work
because of two factors. First, the focus thus far is on the change metrics themselves, and not so much on creating predictive models using those metrics. Second, all the data is derived from a single transformation task implemented one time per language, so there may not be enough data to form useful predictive models and also validate them.

This chapter focuses on the Business Understanding phase of the process, as context is provided to support the other phases as necessary. This chapter, and to some extent Chapter 1, reflects the output of the Business Understanding phase. Later phases of the process are treated in the remaining chapters.

2.1.1 Business Understanding

The “Business Understanding” phase is the preliminary phase of each iteration. Its purpose is to provide or refine a high-level plan for the project and to define success criteria. The plan may include any of the following:

- business objectives, background information specific to the domain, and business-specific success criteria,

- situation assessment; requirements, assumptions, constraints, risks, and contingencies,

- high-level data mining goals and success criteria, and

- assessment of tools and techniques

The term “Business” in this case is used loosely; the CRISP-DM methodology can be adapted in for a wide range of applications, including scientific and engineering
2.1. DATA MINING FRAMEWORK

Figure 2.1: CRISP-DM data mining methodology.

projects [32] and so it does not need to refer to business in the commercial sense. In the case of this thesis, the business domain can be considered the field of Model Based Software Engineering. The “customer”, or audience, then consists of people using, researching, and designing modelling languages in an MSBE context. The high-level goal is to contribute insight about modelling language usage, specifically qualities related to maintainability.

The output of the business understanding phase can depend greatly on the iterative nature of the process. After an iteration, it may be necessary to modify or
remove business objectives from the project plan if analysis results or situational data indicate any discrepancies. It may also be necessary to expand the detail of objectives and assessments because of additional information obtained during an iteration of the process. The goals and objectives of this thesis are detailed in 1.

2.1.2 Data Understanding

In the Data Understanding phase, the available data sources are evaluated and characterised. First, any data that needs to be acquired or accessed should be performed here. Exploratory analysis techniques are selected to provide an accurate view of the “shape” of the data. Analysis during this phase might also serve to satisfy objectives specified in the Business Understanding phase. According to the CRISP-DM documentation, analysis techniques applied to the data include but are not limited to:

- descriptive statistical analysis,
- data visualization, and
- preliminary segmentation into classes.

A report of the data collection is provided in Chapter 4, while exploratory analysis is given in Chapter 5.

2.1.3 Data Preparation

Once the preliminary evaluation of the raw data is complete, the data can be pre-processed for further use. The Data Preparation phase involves selecting and transforming data so that it is suitable for input into analysis and modelling techniques.
Examples of pre-processing tasks include:

- cleaning and/or removing errors in the data,
- constructing derived data,
- integrating multiple sources of data, and
- reorganising data

The Data Preparation phase must be performed with consideration for objectives in the Business Understanding phase and limitations identified in the Data Understanding phase. These form the criteria for selecting data to process and determine how to process the selected data. The preparation of data obtained from the implemented transformation task is explained in Chapter 4. Difference models are generated from transformation artifact versions which are extracted from a version repository. Metrics data is then calculated from these difference models.

2.1.4 Modelling

The Modelling phase takes prepared data and performs additional analysis in support of Business Understanding objectives. Due to the close dependency on prepared data, there can be multiple iterations between the Modelling phase and the Data Preparation phase to support different types of analysis.

- segmentation,
- concept descriptions,
- classification (create a model that assigns an attribute with discrete values to a set of data objects),
2.1. DATA MINING FRAMEWORK

- prediction (similar to classification, only the target attribute is continuous instead of discrete), and

- dependency analysis (create a model describing meaningful associations between data items).

Most of the analysis in this thesis is directed toward exploratory segmentation and classification of software change attributes for different sets of metrics.

2.1.5 Evaluation

After creating a model, it should be evaluated to ensure that it satisfies objectives. From an data analysis perspective, evaluation might involve testing the model against a reserved set of test data, or cross-validating the model against some other measure. At a higher level, the data mining results are evaluated for their usefulness in meeting the project objectives. In this research the created data models are relatively simple and this part of evaluation is covered in detail alongside the analysis in Chapter 5, while higher-level evaluation is carried out in Chapter 6.

2.1.6 Deployment

In the deployment phase, observations and results that satisfy the project objectives are summarized. Any “live” models required as deliverables are implemented and integrated into the customer’s processes. For this research, there are no live models or continuing analyses required. The overall thesis can be considered a part of the deployment, with Chapter 6 providing a summary of the work presented here.
2.2 Mining Software Repositories

A major aspect of the base knowledge in the Software Evolution and MSR fields were covered by Lehman and the “Feedback, Evolution, and Software Technology” (FEAST) project [33]. From software case studies, Lehman made several observations about organizational and technical properties of evolving software systems [34]. Lehman et al. also attempt to form models based on evolution metrics [47]. However, metrics are relatively coarse-grained, and at a level of abstraction higher than the language elements that are considered in this research. The reason this thesis considers lower level metrics is because the research for this thesis is attempting to look at language usage and not just effort prediction, and the implementation artifacts in the demonstrated transformation task are much more contained.

More recent work in the field now known as MSR focuses on integrating different sources of historical data linked to the development and usage software systems, and extracting properties and relationships from within these different sources. The properties and relationships are then used to form insights about the software systems. A brief survey of the field is given by Kagdi, Collard, and Maletic [29]. Source code repositories are the predominant data source studied in MSR research, however MSR researchers have also studied a variety of other sources of data. Examples of these other types of sources include mailing lists and issue tracking databases [11, 48]. Insights are usually targeted to assist developers and project managers in maintaining large software systems.
2.3 Measuring Modelling Languages

A framework for the measurement of software systems as related to software quality is proposed in [12]. Boehm develops a taxonomy of software quality attributes and reports on the relationship between those attributes and various software metrics. Along with similar work that appeared during the same time, it forms the foundation for much work on software measurement and MSR. Boehm and others that follow take a holistic approach to software measurement, arguing that the entire development process, including organizational factors, must be considered when evaluating the quality of software systems. Another important work in software measurement is the Goal-Question-Metric (GQM) approach proposed by Basili [8]. The GQM approach attempts to guide the selection of metrics based on high-level business goals so that there is a better chance that the data obtained is relevant for the problem that needs to be solved. These concepts are developed for software systems in general, so they can be applied in the MBSE context.

Some of the earliest work on measurement of software models as understood in a contemporary software modeling context is in the realm of UML. Some possible reasons are:

1. UML represents one of the earliest widespread uses of what could be considered a modelling language in the sense of MBSE as a foundation for CASE tools in the 1990s,

2. UML is a defacto standard established in industry, so there is a large amount of data that can be measured, and
3. UML is a “general-purpose” modelling language, therefore research work measuring UML models can apply to a wide variety of domains.

UML is a method for describing object-oriented systems, so the already established work on object-oriented analysis metrics is relevant for UML. For example, Marchesi exploited this work when proposing some metrics for UML [40]. Baroni and Brito e Abreu adapted a suite of object-oriented analysis metrics called MOOSE to UML, by creating definitions for the metrics in OCL [7]. Genero et al. performed preliminary work defining metrics [21] for UML class diagrams, later looking at validation of metrics for class diagrams [22] and definition and validation of statechart diagrams [20]. Kim and Boldyreff proposed a set of metrics, developed through informal reasoning without validation [31].

In the more specific field of MBSE, the measurement of transformations and transformation language artifacts is gaining more attention. Vignaga [53], has proposed a suite of metrics for the ATL transformation language and van Amstel et al. have investigated metrics for various language including ATL [2, 5, 3]. Bouten similarly defined metrics for the graph-based transformation language Viatra2 and relates those metrics to quality attributes of the transformations. Kapova et al. looked at using static metrics to reason about the maintainability of QVT transformations [30].

2.4 Model Transformation Languages

The breadth of scope of transformation tasks has resulted in a large variety of model transformation languages. Some of them, for example ATL, QVT, and Henshin (formerly EMFTiger) [27, 24, 10], are designed specifically for transformations. Others, for example Kermeta and Xtend [44, 18] are more general. Though they are suited
to perform a variety of tasks, they provide specific support for operating on models and defining mappings. Czarnecki and Helsen proposed an extensive feature map of transformation languages in 2006 [16]. The feature map describes both internal semantic functions and external language functions of a wide variety of model transformation approaches. Extension points are included for future expansion so that the feature map can grow to accommodate new transformation languages. Mens and Van Gorp proposed a preliminary taxonomy of transformation activities which segments the space of transformations by attributes like the type of input and output [42]. The taxonomy also includes more general characteristics such as complexity and preservation properties. The paper additionally suggests a set of desirable functional criteria for model transformation languages.

This section describes the two transformation languages used to implement the transformation task (Chapter 3). The language descriptions are based on the features and characteristics provided in the work by Czarnecki and Mens. In addition to this description, an effort is also made to describe the tooling available to users of the languages as a part of characterizing the “pragmatics” of these languages. It is useful to show the user experience of these transformation languages in practice, since much of the MBSE methodology depends on development processes that enable efficient transformation of models, including the development of said transformations.

The transformation languages considered are: ATL\(^1\) and the mapping tool in the development platform supporting UML-RT called Rational Software Architect (RSA)\(^2\).

---

\(^1\)See ATL’s project website at http://www.eclipse.org/atl

\(^2\)See http://publib.boulder.ibm.com/infocenter/rsasehlp/v7r5m0, “Extending Product Function”
2.4. MODEL TRANSFORMATION LANGUAGES

2.4.1 ATLAS Transformation Language

The ATLAS Transformation Language (ATL) is a hybrid declarative/imperative transformation language that was conceived as part of the foundation of AMMA and also in response to OMG’s Request For Proposals of a standard transformation language specification, Query/View/Transform (QVT) [28]. Perhaps because of its status as an early member of the Eclipse Modelling Project and its initial alignment with the QVT Request for Proposals, it is a popular choice for M2M transformations within the EMF ecosystem. The language features a textual concrete syntax and its semantics are primarily based on the execution of mapping rules. To provide executability, ATL source artifacts are themselves transformed into virtual machine bytecode, represented as Abstract State Machine models.

Transformation Rule Features

- **Domains.** An ATL rule must involve at least two domains; one source and one target. There can be more than one target domain referenced in the rule.
  - **Domain languages.** ATL is able to use any EMF-based metamodel as a domain language. There is additional specialized support for UML which exploits the EMF-based UML2 implementation of Eclipse Modeling Project. Domain languages are annotated in the header of ATL module files so that they can be referenced by a symbolic name. They are associated with the appropriate metamodels in the configuration before executing.
  - **Static modes.** The source domain always executes in an “in” mode, while the target domains always execute in an “out” mode.
– **Dynamic mode restriction.** There is no facility to configure whether the domain can be restricted to either “in” or “out” modes.

– **Body.** The body of ATL rules, like most other parts of ATL, rely heavily on OCL expressions.

  * Variables can be assigned values based on OCL expressions. These variables are part of the “from” and “to” sections of the ATL rules, which relate to the source and target domains, respectively.

  * Matched and lazy matched ATL rules contain one “in” pattern and one “out” pattern. These patterns correspond syntactically to the “from” and “to” sections. Called rules only contain an “out” pattern because they are not matched with source elements but invoked explicitly with parameters.

  * The logic of ATL rules is mostly declarative owing to the use of OCL. However, rules can make use of imperative condition constructs and loop constructs. Rules can also contain a “using” section which allows for imperative programming. The semantics of OCL are extended to support these constructs.

– **Typing.** ATL rules employ syntactic typing based on the domain metamodels that are referenced in the header of the module. The typing functionality is integrated with OCL typing semantics.

- **Syntactic Separation.** ATL rules are syntactically separated by the “from” and “to” clauses. These clauses provide a boundary between source and target domains. Source domains are considered read-only and target domains are considered write-only.
• **Multi-directionality.** ATL rules execute in one direction only; from source to target.

• **Application Conditions.** An ATL rule can contain an application condition in the “from” clause. The OCL expression that specifies which source element will be transformed can contain additional expressions and control flow constructs.

• **Intermediate Structures.** Each rule has its own scope, where OCL variable expressions can hold intermediate data about transformation elements. Additionally, the ATL engine tracks internal structures such as trace data.

• **Parameterization.** ATL allows special “called” rules that can take arbitrary parameters. This type of rule must be called explicitly from another rule (i.e., it will never be implicitly selected by the matching algorithm).

• **Reflection.** ATL provides access to the rules and OCL helpers in a module using a reflection API within the OCL expression language.

• **Aspects.** There is no support for aspects in ATL rules.

**Rule Application Control Features**

• **Location determination.** The application of rules based on source elements is deterministic, since regular matched rules can only be applied once for a source element. The transformation is considered invalid and an exception is thrown if multiple matched rules satisfy application conditions for a single source element. The two other types of rules (lazy rules and called rules) must be invoked explicitly from within other rules, so their application is based on the navigation
of the source elements. These two cases are not necessarily deterministic because the rules are invoked in the context of OCL expressions which themselves may not be deterministic. The location determination of target elements utilises the traceability links maintained by the execution engine.

- **Rule scheduling.** The scheduling of ATL rules is explicit and internal. Scheduling is explicit in the sense that rules can be invoked by other rules within OCL expressions. Scheduling is internal since the scheduling mechanism is not a separate construct of the language, but is rather a result of call order in the OCL expressions. The concept of scheduling only makes sense for called rules and lazy rules, since there is necessarily a one-to-one relationship between matched rules and source elements. Because there is no matching for called and lazy rules, the selection of these rules is by their name. The engine recursively iterates through the rules. Execution semantics according to [28] show two phases in the transformation algorithm. In the first phase, a match phase creates traceability links for all the matched rules. In the second phase, the algorithm creates and initializes the target elements using their binding expressions. This second phase also includes the scheduling of called and lazy rules as the binding expressions are evaluated.

**Rule Organization Features**

ATL transformations can be organized into modules. At execution time, ATL modules can be combined using a concept called “superimposition”, where the transformation rules are amalgamated into a single transformation module. ATL transformation rules can also use a built-in inheritance mechanism for reuse. The matched rules are
organized based on source elements because of their one-to-one relationship. However called rules and lazy rules are independently organized since they have a less restricted relation to source elements.

**Source-Target Relationship Features**

ATL transformations require that the source model and the target model are separate, since the source model is read-only and the target model is write-only. However the transformations can be configured at execution time to simulate in-place transformations. The simulation is performed by taking the resulting target model and automatically merging it with the source model.

**Incrementality Features**

ATL transformations execute as a whole and the current rule matching algorithm of the execution engine does not allow incremental partial updates to the target. The engine also does not use previously created traceability links to detect incremental updates based on the source model.

**Directionality Features**

Transformations in ATL are one-directional, from source to target. A bi-directional transformation must be implemented by two separate ATL transformations.

**Tracing Features**

ATL features support for tracing in its execution engine. As per [28], each trace contains: a reference to the executed rule, a reference to the source elements,
2.4. MODEL TRANSFORMATION LANGUAGES

a reference to the target elements. Traces are only generated for regular matched rules. The in-memory traceability model is used by ATL’s execution engine when determining how to handle rule matches, however it can also be serialized as an artifact of the executed transformation.

Tools

Figure 2.2 shows the main editor for ATL transformation artifacts. ATL integrates with the Eclipse development platform to provide tooling for transformation developers. Since the concrete syntax is textual, the transformation editor resembles other Eclipse text editors such as Eclipse’s Java source code editor. The executability of ATL source artifacts allows tooling to detect syntactic errors and warnings, displaying them in the standard Eclipse Problems view. The Outline view displays a tree-based representation of the transformation’s structure. This tree is essentially a view of the ATL transformation’s abstract model. Both the Problems view and the Outline view are updated on-the-fly.

Figure 2.3 shows the tools available while executing a transformation. Like the ATL editor perspective, ATL provides extensive integration with the Eclipse development platform. In this case, the Run/Debug configuration UI is used to specify execution options. If a transformation is run without debugging, print output can be shown in a console or piped to a file. If the transformation is executed with debugging enabled, then the Eclipse Debug perspective can be used to examine the running transformation, as shown in Figure 2.3. The debugger can interactively step through the lines of code in the ATL transformation source, while showing the context of the transformation at the point of execution. “Context” in this case means the source
and target model elements that are defined up to the point of execution in the rule that is being executed. This context is presented in the Variables view. The ATL tooling also allows transformation developers to set conditional breakpoints and to query the context using OCL.

One issue with the Variables view is that model elements are represented as trees, where inherited classes are shown as nodes in the tree. One needs to drill down to see all the element’s features. For model elements with large inheritance trees, this makes the Variables view awkward to use.
2.4. MODEL TRANSFORMATION LANGUAGES

Figure 2.3: Eclipse Debug perspective showing a paused execution of an ATL transformation. The current execution position is shown in the code editor (bottom), and the context of the transformation at that execution position is shown in the Variable view (top-right).

2.4.2 RSA Mapping Language

Rational Software Architect (RSA) is one of IBM’s solutions for the software modelling process. This solution is an extension of the Eclipse platform and it takes advantage of EMF. The RSA mapping tool provides a graphical representation of transformation mappings and a Java framework to support features such as custom rules, filters, and conditions. The execution semantics support a top-down hierarchical approach to model transformations.
The Java framework represents a superset of the model transformation capabilities that are included with RSA. In addition to the mapping engine, there is also support for different model merging strategies, custom Java-based transformations with different rule application strategies, and many (though sparsely documented) extension hooks for most aspects of the transformation process.

Transformation Rule Features

• **Domains.** Mapping rules in the RSA mapping tool require two domains; a source domain and a target domain.
  
  – *Domain languages.* Domain languages are specified under the root of the mapping model file. The user can select any metamodel or UML profile that is registered in the RSA environment. The mapping model file contains two separate lists of input and output domain languages.

  – *Static modes.* The source domain corresponds to an “in” mode, while the target domain corresponds to an “out” mode. All RSA mapping rules are unidirectional, so the “in/out” mode does not apply.

  – *Dynamic mode restriction.* There is also no dynamic rule restriction concept at the mapping rule level, although during configuration and execution of the transformation, users can specify whether or not auxiliary output models will persist.

• **Body.**

  * The body of mapping rules do not explicitly support the declaration of variables, but internal Java code referenced by the rules (in the form of method bodies) can create local variables.
2.4. MODEL TRANSFORMATION LANGUAGES

* Patterns are specified via the graph-based abstract structure of the source and target metamodels. The depth of mapping rule patterns is limited to a single level below the mapped elements. In other words, mapping rules do not contain nested rules; the transformation developer must first create a mapping containing the deeper rules and refer to it using a submap rule.

* A Java framework underlies the RSA mapping tool, so the logic of mapping rule bodies are specified imperatively using plain Java. Java code is specified either internally as method bodies that are injected into the generated transformation implementation, or externally as Java classes that implement framework-specific interfaces.

• **Syntactic separation.** The mapping tool features partial syntactic separation, in that certain parts of the mappings such as conditions and filters only allow access to one domain or the other. For example, a Java method body that implements a rule condition will only be given the mapping’s source element as a parameter. However, when implementing a custom rule, there is no such restriction on the access of domains; both the source element and the target element will be passed to the custom rule code.

• **Multi-directionality.** There is no support for multidirectionality at the rule level.

• **Application conditions.** Mapping rules feature conditions that are specified in Java, interacting with the RSA transformation framework. The mapping rule conditions are implemented in one of two ways; internal to the mapping
model file or externally. External conditions are Java classes that implement the “RuleCondition” interface. The fully-qualified class name is then referenced in the mapping file.

- **Intermediate Structures.** During execution, each mapping is given its own transformation context, essentially a map of properties. The immediate context of the mapping is combined with the contexts of all ancestor mappings in the execution hierarchy. Internally, the transformation framework uses this context to track state and to manage trace links between the source and target models. Transformation implementers can also access this context. The transformation framework will detect whether external Java classes implement the ContextExtension interface and it will pass in the context by calling the method “getContext” in the interface that is implemented by the external Java class.

- **Parameterization.** There is no explicit support for parameterization in the rules or mappings. Users can access the transformation context as detailed in the “Intermediate Structures” section. However, the scoping and management of properties in the context can become confusing because of the large number of objects stored in a flat data structure and because of the presence of internal properties that should only be accessed by the transformation engine.

- **Reflection.** The mapping tool does not expose any concept of reflection. The underlying transformation framework is exposed in custom rules, and it can be manipulated there.

- **Aspects.** The mapping tool does not have a facility to specify mapping rule aspects.
Rule Application Control Features

- **Location determination.** Mapping rules are applied deterministically in a top-down fashion, beginning with the first mapping and the root element of the source model. The location of the target in the first mapping is also deterministic, starting with the root of the target model.

- **Rule scheduling.** Rule execution in the mappings are scheduled explicitly and externally. There are two external mechanisms that combine for rule scheduling; rule sequence and submap rules. The execution of rules in a mapping are performed in the sequence that is specified in the mapping model file. This sequence is editable using the tool. Submap rules can be used to invoke other mappings with additional rules. Only a single rule is executed if multiple rules apply to the same source element. In this case the first rule in the sequence that satisfies the rule condition will be executed. There is no concept of phasing, just a single rule execution phase in which the entire transformation is performed.

Rule Organization Features

The RSA mapping tool allows users to reference external mapping model files and use the mappings contained in those external models. The organizational structure of rules is independent of either the source or the target metamodels. Mappings can be reused via an inheritance/specialization mechanism.

Source-Target Relationship Features

Source and target are considered separately when executing the mapping transformation. In-place transformations can be simulated by configuring the transformation in
“Mixed” mode.

**Incrementality Features**

There is no concept of incrementality in the way mappings are executed by the transformation framework. Transformations are executed in whole, however the results are optionally reconciled with the pre-execution source and target models.

**Directionality Features**

The RSA model transformation framework provides the option of referencing a “reverse” transformation. This is intended to be used for the “Reconciled” design contract management protocol. However, the reverse transformation must be implemented manually.

**Tracing Features**

There is an option in the transformation configuration file to generate a log file containing rule execution traces. The log is an XML file containing the traces of all mapped elements in the order that the mappings were executed. In addition to execution order, the hierarchical structure of nested mappings is preserved. This trace file is primarily intended for debugging purposes. During execution, traceability links are stored in memory by the framework and used to determine how rules should interact with the source and target models. The creation of internal traceability links is fully automatic and there is no direct way for users to configure how they are created.
Tools

Figure 2.4 shows a screen shot of the graphical mapping tool. The view represents a single mapping declaration. The source element class is shown on the left side of the view, while the target element class is shown on the right. Rules, shown as boxes in the middle connecting source and target features, provide a way to refine the mapping declaration, via either a submap reference or a custom Java helper method.

On the right hand side, the outline view shows a list of all the mappings in the mapping model. The ordering of the mappings can be changed using the context menu, however there is no multiple selection functionality in this view so only one mapping or rule can be moved at a time. Further, the outline view is not capable of drag-and-drop interaction, limiting move operations to one position in either direction.

Mapping transformations are invoked using the tool integration provided by RSA’s transformation framework. The user must first create a transformation configuration file and reference the desired transformation. Figure 2.5 shows the default view of a newly created transformation configuration. The protocol option governs how the source-target relationship of the transformation is managed by the framework. In the default “Conceptual” mode, the target model is kept separate from the source model. In “Mixed” mode, elements in the target model are incorporated in the source model. Finally, in “Reconciled” mode, an inverse transformation is applied to the target model and then compared to the original source model. “Reconciled” mode assists in synchronizing the source model to the target model.
The option to generate a debug log relates to the traceability feature of the framework. Support for traditional interactive debugging is delegated to the default debugging facility of RSA/Eclipse Java Development Tools. This is possible because execution of RSA transformations is handled completely by the Java-based transformation framework. The drawback of this approach is that the low-level details of the framework are exposed to the user, potentially causing confusion. Further, it is not easy to step between successive mapping rules because of the architecture of the framework.

RSA’s transformation framework provides model loading capabilities for a variety of model representations including UML models and EMF models. The model representation types for source and target models are specified during the implementation.
2.4. MODEL TRANSFORMATION LANGUAGES

Figure 2.5: The main panel of a transformation configuration used to invoke an RSA mapping.

of the transformation, and this determines how models are selected in the transformation configuration options when executing the transformation. The interface that enables this is shown in Figure 2.6.
Figure 2.6: The "Source and Target" panel of a mapping transformation configuration. This panel allows the user to select the input and output models for which the transformation will execute.
2.5 Related Work

There already exist qualitative comparisons of languages, for example [23] comparing the transformation languages ATL, QVT, and XText. Schubert compares and evaluates ATL, Epsilon Transformation Language, QVT, and Xtend [50].

Amstel et al. have published work that analyses, using metrics, various quality attributes of model transformation implementations in SDF+ADF [1] and ATL [4]. To evaluate quality and correlate quality attributes to metrics, the authors rely on expert opinion. This differs from the approach in the research here; this thesis work attempts to analyse historical language usage data to evaluate qualities of model transformation artifacts and languages. The work here can be viewed as complementary to the work in [1], since it can be used to further support the correlation of metrics to quality properties. Exploratory data mining can also help researchers decide which metrics to use.

Some researchers have exploited the abstract syntax of code artifacts to perform analysis, for example Maletic et al. [38]. Like their work, this thesis uses abstract syntactic information to represent implementation artifacts and to produce differencing information. However Maletic et al. place more emphasis on the traceability of differences back to original text-based artifacts, and they generally focus on the differencing approach.

In terms of analysing and mining change data in MBSE, there exists work on co-evolution and change impact analysis of model artifacts. For example, Ivkovic and Kontogiannis propose a change model to support traceability and synchronization of changes across model artifacts [26]. Briand, Labiche, and O’Sullivan use version differencing to perform an impact analysis on UML models [13]. There also exists some
work to apply mining techniques to historical model repositories [55], but that work is more concerned with the high-level model-driven process and process management. Perhaps the most closely related work to this thesis is that of Lehnert, Farooq, and Riebisch, who propose a change type taxonomy [35]. In contrast to the research in this thesis, Lehnert et al. define their own more general graph-based change model, and then they map their change model to the Ecore metamodel. The main long-term goal of their research is to support change impact analysis of EMF-based models.
Chapter 3

Transformation Task

The demonstrated transformation task in this thesis is an exogenous model transformation. The transformation translates model instances of UML-RT, a modeling language of real-time software systems, into model instances of kiltera [46], a modeling language based on process algebra. For the purpose of this research, we only consider the subset of the transformation that converts UML-RT state machines into kiltera process definitions. The full scope of the mapping is much more extensive and includes many other features of the UML-RT language, such as capsule (active class) structure, thread-aware deployment, dynamic port binding, and simplified action language support. The transformation was developed in the context of an ongoing collaboration with IBM Canada [17] aimed at improving the execution and analysis support for UML-RT and was created over several iterations, as more features from UML-RT were translated into kiltera code. The part of the mapping that is relevant to this thesis work is described in a technical report [45]. The technical report provides a mathematical specification for our transformation task. The remainder of this chapter will refer to the technical report as the “mapping specification”.
3.1 Transformation Process

The overall process of the transformation that implements the mapping specification is shown in Figure 3.1. The inputs and outputs of the main mapping transformation are a model of a UML-RT system and a corresponding model of a kiltera program, respectively. A pre-processing model-to-model transformation selects the relevant elements of the UML-RT model and converts them into the intermediate simplified representation. This intermediate representation is described in more detail in the next section. A post-processing model-to-text transformation converts the kiltera program model into textual kiltera code.

3.2 Transformation Inputs and Outputs

Though the input and output mathematical syntaxes were mostly defined at the start, minor changes were made to reflect misunderstandings regarding the UML-RT models and requirements for more language constructs in the kiltera models. In the transformation task for this research, the metamodels are deliberately fixed throughout the evolution of the mapping specification. The intent of the fixed metamodels is to simplify and reduce the number of variables in the experiment.
3.2. TRANSFORMATION INPUTS AND OUTPUTS

The input metamodel of the UML-RT system, over which input models are typed, is actually a partial representation of the entire UML-RT metamodel. There are three primary reasons guiding the decision to use this intermediate metamodel:

1. the mapping specification defines and uses a mathematical syntax for UML-RT so it makes sense to represent inputs to the mapping in a format that is close to this syntax,

2. some of the properties of input models specific to UML-RT are accessed with an API provided by a proprietary RSA plugin which limits the use of UML-RT models to the RSA environment, and

3. the UML 2.0 metamodel upon which UML-RT is based is complicated, so a simplified input metamodel helps to reduce complexity of the implemented transformation.

Similarly, the piklt metamodel represents a subset of the full kiltera language; it contains only the kiltera language constructs used by the mapping specification. Also, the concrete syntax of piklt is more compact than that of the kiltera language and it is better suited for the style of mathematical definitions in the specification. An Ecore metamodel for piklt is employed in the transformation implementation so that the implementation as a whole corresponds more directly to the mapping specification.

3.2.1 UML-RT state machine syntax

UML-RT is defined as an extension of UML 2 using the UML profile mechanism. UML-RT state machines are therefore heavily derived from UML 2 state charts. UML-RT state machines specify the behaviour of capsule instances, the active objects in a
3.2. TRANSFORMATION INPUTS AND OUTPUTS

UML-RT system. State machines are restricted to a single region. Figure 3.2 shows a simple example of the state machine in a UML-RT capsule.

Using the mathematical syntax defined by the mapping specification, the example in Figure 3.2 is represented by the following tuples:

- **State tuples**:

  \[
  s_0 := [n_0, \{\text{den}\_n0\}, \{\text{dex}\_n0\}, \{s_1, s_3\}, 1, \langle \text{init}\_n0, t_1 \rangle, \bot, \bot]
  \]

  \[
  s_1 := [n_1, \{\text{den}\_n1, \text{en}\_1\}, \{\text{dex}\_n1, \text{ex}\_1\}, \{s_2\}, 1, \langle \text{init}\_n1 \rangle, \bot, \bot]
  \]

  \[
  s_2 := [n_2, \{\text{den}\_n2\}, \{\text{dex}\_n2\}, \bot, \bot]
  \]

  \[
  s_3 := [n_3, \{\text{den}\_n3\}, \{\text{dex}\_n3\}, \bot, \bot]
  \]

- **Transition tuples**:

  \[
  \text{init}\_n0 := (\text{in}, \text{true}, \bot, \text{en}_1, \bot, \bot)
  \]

  \[
  \text{init}\_n1 := (\text{in}, \text{true}, \bot, \text{den}\_n2, \bot, \bot)
  \]

  \[
  t_1 := (\text{sib}, \text{true}, \text{ex}\_1, \text{den}\_n3, \text{port1}\_\text{ping}, \bot)
  \]
Tuples \( s_0, s_1, s_2, \) and \( s_3 \) define states in the model, while tuples \( \text{init}_0, \text{init}_1, \) and \( t_1 \) define transitions. Sets are denoted using curly braces (\{\ldots\}) while sequences are denoted with angle brackets (\langle\ldots\rangle). Tuples that define basic states contain, in order: the name, a set of entry points, a set of exit points, the entry actions, and the exit actions. The representation of composite states extends the tuples to include a set of contained states, an index representing the default transition, and a sequence of contained transitions (the entry and exit actions are always last for both basic and composite cases). Tuples that define transitions contain the transition ‘kind’ (one of ‘sibling’, ‘in’, or ‘out’), a boolean indicating whether the transition is first in its chain, the source, the target, the trigger event, and actions. Transition sources and targets can be both entry points and exit points. Null triggers and actions are denoted by the symbol: \( \bot \). Note that though \( n_0 \) is technically a UML state machine in the UML-RT model, it is represented as a state in the UML-RT mathematical syntax. Further details of the UML-RT mathematical syntax can be found in the mapping specification.

The mathematical syntax is implemented as an Ecore metamodel plugin in EMF. Figure 3.3 shows the relations between classifiers describing state structure, while Figure 3.4 shows the relations of classifiers for transitions. All ‘nodes’ that can exist in a StateMachine are specializations of the Vertex class. This allows transitions to use different types of nodes as sources and targets. The views of the metamodel shown here are restricted to the parts that are relevant for Milestones 1 to 5.

As shown in Figure 3.4, a Transition can have an Action associated with it. This corresponds to the action code of UML-RT transitions. This Action classifier can be specialized by other metamodels in order to provide semantics for different action
Figure 3.3: A view of the state structure classes in the simplified UML-RT metamodel.

languages. For example, the mapping specification defines a simplified action language is defined in a separate Ecore metamodel in order to support basic functions of the UML-RT run-time framework like timer management and capsule role lifecycle management. The Vertex classifier’s “outgoingTransitions” association assists in navigating chains of transitions that can be connected by Vertex instances.

Figure 3.4: A view of the classes related to transitions in the simplified UML-RT metamodel, as used in the transformation task.
3.2.2 kiltera syntax

The language kiltera\(^1\) is a language that models concurrent, distributed, mobile and timed processes. The language is based on a process algebra called piklt, has formal semantics and a rich meta theory including timed notions of behavioural equivalence.

![Diagram of kiltera process constructs](image)

(a) Basic concurrency, output, and initialization processes

![Diagram of encapsulation and named composition constructs](image)

(b) Encapsulation and named composition constructs

Figure 3.5: Basic kiltera process constructs (a), and basic kiltera program structure constructs (b)

Like the simplified UML-RT syntax, piklt is implemented as an Ecore metamodel

\(^1\)Available at http://www.kiltera.org
so that its abstract syntax may be used in the transformation. The mapping specification introduces this syntax for piklt and refers to it in mapping specification definitions. Processes are the basic entities that comprise a piklt program. A process can be considered a unit of execution. Processes can be combined in various ways, including sequentially and concurrently, enabling the expression of non-determinism. The Ecore metamodel classes that correspond to this basic example are shown in Figure 3.5. Consider the following example:

```piklt
proc BasicExample() = def {
    proc A() = print "x"; √
    proc B() = print "y"; print "z"
} in A()||B();
```

In the preceding example, the named process definition BasicExample contains a local definition process (denoted as “def”). This local definition houses two additional process definitions named A and B.

- Process A is a sequential process denoted by the operator “;”, which first prints the output “x” and then executes the trivial null process denoted by “√”.

- Process B is another sequential process that executes two subprocesses, outputting “y” and “z” in order.

The local definition process then starts a parallel process, denoted by the operator “||”, which creates and executes instances of the process definitions A and B. These
process instances execute non-deterministically in parallel. Due to non-determinism, there are three possible outputs for this example program:

1. “x”, “y”, “z”,
2. “y”, “z”, “x”, or
3. “y”, ”x”, ”z”.

Processes in kiltera use message passing as their main communication mechanism. Message data associated with events can be matched with patterns at the receiving end. There are also additional helper constructs that support control flow such as conditionals using boolean expressions. The following example demonstrates these constructs:

```plaintext
proc MessageExample() = def {
  proc BinaryOpInput(x,a,b) = x!(a,b);
  proc DivideOp(x) = {
    x?(dividend,divisor).
    if divisor = 0 then print ”undefined”.
    else print dividend/divisor.
  }
  new x in BinaryOpInput(x,4,2)||DivideOp(x);
}
```
In this example, an instance of the BinaryOpInput process is executed in parallel with an instance of the DivideOp process. Both process instances are passed the event “x” over which communication between the two processes will occur. Additionally, the instance of BinaryOpInput is passed the parameter values 4 and 2. When the BinaryOpInput process triggers event x with the two values in tuple form “(4, 2)”, the event listener in the DivideOp will perform a pattern match and use the values in the tuple as dividend and divisor. The event listener then triggers a conditional branch process that outputs “undefined” if the divisor is zero. Otherwise it will perform the division and output the result.

Triggered events are “lasting”, in other words a triggered event will persist even if no listener exists to handle it yet. Therefore this example will always terminate with the output “2”. This example also illustrates the use of parameters bound as input when creating process instances.

The Ecore metamodel classes that represent message passing and control flow, some of which are introduced in this second example, are shown in Figure 3.6.

### 3.3 Transformation Specification Milestones

The mapping specification is defined incrementally in terms of UML-RT features. The steps required to implement milestone \( n \) given an implementation of milestone \( n - 1 \) are defined as a set of changes. Figure 3.7 shows how the transformation grows in terms of milestones. This section provides an overview of the change steps for the milestones of the transformation task. In MSR-related work, the detection of change “transactions” is sometimes based on commit time data or issue tracking data.

For each milestone, a brief description of implemented features is included along
3.3. TRANSFORMATION SPECIFICATION MILESTONES

(a) kiltera processes that handle message passing between processes and the creation of events.

(b) kiltera processes that provide means for branching, pattern matching, and timing during execution.

Figure 3.6: kiltera processes for data flow (a) and control flow (b).

with a table of the change steps. Milestone 1 is the initial milestone, therefore it does not contain a table of change steps. Additionally, the descriptions of milestones 1 and 2 include their mapping definitions as they appear in the specification, and the first table of change steps is explained in detail. The mapping definitions of subsequent milestones are not included, and instead are referenced in the specification for the sake of brevity.
3.3. TRANSFORMATION SPECIFICATION MILESTONES

Figure 3.7: The incremental development of UML-RT language features defined in the transformation’s mapping specification, by milestone.

3.3.1 Milestone 1: hierarchical state structure

The initial milestone of the mapping specification describes the semantics of hierarchical nesting of states in UML-RT state machines as well as simplified semantics for initial points and initial transitions. State machines can have a set of children states, and the executing state machine can be “in” one of its children states. State machines can also contain an initial point and transition leading to a default child state. Finally, each state is itself a state machine, enabling the hierarchical nesting of states.

In the example shown in Figure 3.8, primary state machine $n0$ contains one state, $n1$. State $n1$ contains two states, $n2$ and $n3$. Upon execution, the primary state machine takes the transition $init_{n0}$ from the initial point and enters state $n1$. When state $n1$ is entered, the transition $init_{n1}$ executes from the initial point and enters state $n2$.

The definitions in Figure 3.9 taken from the mapping specification [45, p.14] describe how model instances defined in the UML-RT syntax are mapped to kiltera
3.3. TRANSFORMATION SPECIFICATION MILESTONES

Figure 3.8: UML-RT state machine example for Milestone 1: states n2 and n3 are nested within state n1, the initial transition of state n0 targets state n1, and the initial transition of state n0 targets state n2.

models in the piklt syntax. Definition $\mathcal{T}_0$ describes a mapping of UML-RT states to kiltera process definitions. Helper functions such as $\text{name}(...)$ are defined in the specification alongside the input and output syntaxes.

\[
\mathcal{T}_0[[n, A, B, en, ex]] \overset{def}{=} \text{proc } S_n() = \sqrt{ }
\]

\[
\mathcal{T}_0[[n, A, B, S, d, T, en, ex]] \overset{def}{=} \text{proc } S_n() = \text{def } \{D_1; \ldots; D_k\} \text{ in } S_{n,d}()
\]

where each $D_i$ is $\mathcal{T}_0[s_i]$ for each $s_i$ in $S = s_{1..k}$, and $n_d = \text{name}(s_d)$ is the name of the default sub-state.

Figure 3.9: An excerpt of the specification containing the mapping definition that corresponds to Milestone 1.
The representation in the mathematical syntax for the example in Figure 3.8 is:

\[ s_0 := [n_0, \{den_n0\}, \{dex_n0\}, \{s1\}, 1, \langle init_n0\rangle, \bot, \bot] \]
\[ s_1 := [n_1, \{den_n1\}, \{dex_n1\}, \{s2, s3\}, 1, \langle init_n1\rangle, \bot, \bot] \]
\[ s_2 := [n_2, \{den_n2\}, \{dex_n2\}, \bot, \bot] \]
\[ s_3 := [n_3, \{den_n3\}, \{dex_n3\}, \bot, \bot] \]
\[ init_n0 := (in, true, \bot, \bot, \bot) \]
\[ init_n1 := (in, true, \bot, \bot, \bot) \]

After transforming the example using the mapping definition for Milestone 1, the result in mathematical kiltera syntax is:

```
proc S_n0() = def {
    proc S_n1() = def {
        proc S_n2() = \checkmark;
        proc S_n3() = \checkmark;
    } in S_n2();
} in S_n1();
```

Table 3.1 shows the changes required to implement the first milestone. A simple null process represents basic states, while nested process definitions represent composite states.
Table 3.1: Change steps for the completion of Milestone 1

### Milestone 2: basic transitions

Basic transitions, according to the mapping specification, are transitions of the type ‘sibling’, for which the source is a basic state. In the example illustrated in Figure 3.10, t1 is an instance of a basic transition. After the capsule state machine initializes, it waits in state n1. When transition t1 is triggered, then the state machine enters its target state n2.

Figure 3.10: UML-RT state machine example for Milestone 2: t1 is a ‘basic’ transition

Representing the example in Figure 3.10 using the mathematical syntax results in the following:

\[
\begin{align*}
    s_0 & := [n_0, \{den\_n0\}, \{dex\_n0\}, \{s1, s2\}, 1, \langle init\_n0, t1 \rangle, \perp, \perp] \\
    s_1 & := [n_1, \{den\_n1\}, \{dex\_n1\}, \perp, \perp] \\
    s_2 & := [n_2, \{den\_n2\}, \{dex\_n2\}, \perp, \perp] \\
    init\_n0 & := (in, true, \perp, den\_n1, \perp, \perp) \\
    t1 & := (sib, true, dex\_n1, den\_n2, port1\_ping, \perp)
\end{align*}
\]
3.3. TRANSFORMATION SPECIFICATION MILESTONES

Figure 3.11 shows the mathematical mapping for the feature set of Milestone 2, as it is listed in the mapping specification [45, p. 15-16]. The definitions for mapping \( T_1 \) build upon \( T_0 \). Instead of being represented by the null process, each basic state is mapped to an event handling process that matches on transitions that exit the state. This set of transitions is defined by \( T'' \). The mapping for composite states remains the same, since by definition “basic” transitions cannot originate from composite states.

<table>
<thead>
<tr>
<th>Definition 5. (Encoding basic transitions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>• For a basic state ( s \equiv [n, A, B, en, ex] ) its translation is given by:</td>
</tr>
</tbody>
</table>
| \[
T_1[s], s \vdash \text{proc } Sn() = \sum_{t_i \in T'} x_i.\text{Sn}_i()
\]
| where \( s \)'s enclosing state is |
| \( s' = [n', A', B', S', d', T', en', ex'] \) |
| and |
| \( T'' \equiv \{ t \in T' \mid \exists q \in B. q = \text{src}(t) \} \) |
| is the set of transitions from \( T' \) whose source \( (q) \) is an exit point of state \( n; \) |
| \( x_i \equiv \text{ev}(t_i) \) |
| is the trigger event of transition \( t_i \) in the set \( T'' \); and |
| \( n_i = \text{name}(s_i) \) |
| is the name of the target state \( s_i \in S' \) with \( \text{trg}(t_i) \in \text{entries}(s_i) \) (the target of the transition must be an entry point of the target state \( s_i \)). |

• For a composite state \( s = [n, A, B, S, d, T, en, ex] \) (with enclosing state \( s' \)): |
| \[
T_1[s], s \vdash \text{proc } Sn() = \text{def } \{D_1; \ldots; D_k\} \text{ in } Sn_d()
\]
| where each \( D_i \) is \( T_1[s_i] \), for each \( s_i \) in \( S = s_{1..k} \) and \( n_d = \text{name}(s_d) \). Note that in this case the parameter passed to the translation of the sub-states is \( s \), the composite state being translated. It is not necessary to pass the enclosing state \( s' \), since in UML-RT, transitions cannot cross boundaries. The state \( s' \) will be used later when we deal with group transitions. |

Figure 3.11: An excerpt of the specification that shows the mapping definition of Milestone 2.
The application of this mapping to the example shown in Figure 3.10 results in the following piklt model:

\[
\text{proc } S_n0() = \text{def } \{
\text{proc } S_n1() = \text{port}_1\text{ping}?:S_n2();
\text{proc } S_n2() = \sqrt{;}
\} \text{ in } S_n1();
\]

When an instance of the primary state machine process \( S_n0 \) is executed, the state process \( S_n1 \) will in turn be executed. This will cause the program to listen for an event \( \text{port}_1\text{ping} \). If an event \( \text{port}_1\text{ping} \) is triggered, then an instance of process \( S_n2 \) will execute and the process \( S_n1() \) will terminate. The instantiation of a state process is considered equivalent to entering the state, while the termination of a state process instance is considered equivalent to leaving that state.

Note that this mapping will depend on an environment that defines the event \( \text{port}_1\text{ping} \), which is based on capsule ports and their corresponding protocols. These UML-RT features are introduced in a later version of the mapping specification document, where capsule semantics are defined. Also note that the process \( S_n2 \) will terminate immediately when it is executed, since there are no further possibilities for transition triggers. This behaviour differs from the actual UML-RT run-time, since the framework does not terminate the state machine in this situation. The discrepancy is addressed in later milestones of the specifications, when state machine group transition priority semantics are mapped to kiltera. This requires that each state contains an event handler for an “exit” event even if there are no possible transition
trigger events.

This situation is present in the tech report because the tech report is in one sense a historical account of the development of the UML-RT semantics. Each milestone includes a self-contained mapping and the next milestone refines that mapping and addresses its shortcomings.

In order to implement the additional features defined by Milestone 2 of the specification, the refinement process is broken down into change steps. Each change step is comprised of 4 values which form an identifiable unit of change from a part of the original milestone to a part of the target milestone. Table 3.2 shows the change steps required to evolve from Milestone 1 to Milestone 2. The changes add an event handler (2.4) to the basic states in the state machine to capture trigger events of basic transitions.

<table>
<thead>
<tr>
<th>Id</th>
<th>Description</th>
<th>M1 Context</th>
<th>M2 Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Define “transition target state to kiltera name” mapping</td>
<td>n/a</td>
<td>1.11: (n_i = \text{name}(s_i))</td>
</tr>
<tr>
<td>2.2</td>
<td>Define “transition trigger event to kiltera name” mapping</td>
<td>n/a</td>
<td>1.9: (x_i := \text{evt}(t_i))</td>
</tr>
<tr>
<td>2.3</td>
<td>Define set of transitions originating from a given basic state</td>
<td>n/a</td>
<td>1.6: (T'' := { t \in T'</td>
</tr>
<tr>
<td>2.4</td>
<td>Replace basic state null process with trigger event handler</td>
<td>1.2: proc (S_n() = \sqrt{\ldots})</td>
<td>1.2: proc (S_n() = \sum_{t_i \in T''} x_i?.S_{n_i}())</td>
</tr>
</tbody>
</table>

Table 3.2: Change steps from Milestone 1 to Milestone 2

The numeric identifier is defined to be \(x.y\), where \(x\) is the milestone number and \(y\) is the number of the change that determines where in the sequence that change should be implemented. Each change step also contains a natural language functional description of the change. Finally, a change step describes two contexts, one for the “before” milestone and one for the “after” milestone. A context begins with a line
number or a range of line numbers, referencing the change within the milestone’s definitions. Line numbers are derived for the mapping definitions as they appear in the mapping specification text. Prose and top-level equations are counted together starting from 1. If the context must reference a location within a multi-line sub-equation, a secondary line number will be appended, separated by a decimal. This secondary line number will start from 1, and it will count the lines of the sub-equations. Finally, the context includes a relevant excerpt of the mapping definition where the change takes place. Together, a pair of “before” and “after” contexts allows the change step to map between unique positions in the “before” and “after” milestones.

3.3.3 Milestone 3: entry points and incoming transitions

Figure 3.12: UML-RT state machine example for Milestone 3: the execution of transitions t1 and t3 continue along different paths inside state n2 because they target different entry points

Entry points provide a way to direct “incoming” transitions to different targets. These so called “incoming” transitions originate from a state boundary and target a vertex within that state. Consider the example shown in Figure 3.12. In this example, transitions t2 and t4 are “incoming” transitions targeting states n3 and n4 inside state n2, respectively. Depending on whether transition t1 or transition t2 is
triggered, the final target state is different.

The change steps that are used to evolve from Milestone 2 to Milestone 3 are given in Table 3.3. Milestone 3 adds a dispatcher process (3.3, 3.5) that is executed during the initialization of a composite state process (3.6). Instances of composite state processes are passed a parameter (3.4) identifying the entry point through which the state is entered. The dispatcher uses this parameter to execute the target connected to the entry point.

<table>
<thead>
<tr>
<th>Id</th>
<th>Description</th>
<th>M2 Context</th>
<th>M3 Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Add entry point parameter “enp” to “basic state” process definition</td>
<td>1.2: ... proc $S_n() = \ldots$</td>
<td>1.2: ... proc $S_n(enp) = \ldots$</td>
</tr>
<tr>
<td>3.2</td>
<td>Add entry point parameter to process instantiation in “basic state handler”</td>
<td>1.2: ... $x_i?S_{ni}()$</td>
<td>1.2: ... $x_i?S_{ni}(p_i)$</td>
</tr>
<tr>
<td>3.3</td>
<td>Add “entry point dispatcher” process definition</td>
<td>n/a</td>
<td>1.20-25: proc $C(enp) = \ldots$ else $S_{ni}(p_i')$</td>
</tr>
<tr>
<td>3.4</td>
<td>Add entry point parameter to “composite state” process definition</td>
<td>1.15: ... proc $S_n() = \ldots$</td>
<td>1.17: ... proc $S_n(enp) = \ldots$</td>
</tr>
<tr>
<td>3.5</td>
<td>Insert “entry point dispatcher” process definition in “composite state” local definition</td>
<td>1.15: ... def { $D_1; \ldots; D_k$ } $\ldots$</td>
<td>1.17: ... def { $D_1; \ldots; D_k; C_{def}$ } $\ldots$</td>
</tr>
<tr>
<td>3.6</td>
<td>Replace “default state” process instantiation with “entry point dispatcher” process instantiation</td>
<td>1.15: ... in $S_{ni}()$</td>
<td>1.17: ... in $C(enp)$</td>
</tr>
</tbody>
</table>

Table 3.3: Change steps from Milestone 2 to Milestone 3

### 3.3.4 Milestone 4: exit points

Similar to the function of entry points as defined in Milestone 3, exit points provide a mechanism for “outgoing” transitions to follow different paths out of their state. The example in Figure 3.13 shows how two “outgoing” transitions in the same state can follow different paths via their respective exit points. After initialization, the state machine waits in state n2. If the environment inputs a signal “port_ping” then
transition $t_1$ will be taken and the state machine will follow its exit point through to
$n_3$. Alternatively, if the environment inputs a signal “port-ping2”, then transition $t_3$
will be taken and the state machine will enter state $n_4$ following the other exit point
and transition $t_4$.

![UML-RT state machine example for Milestone 4: transitions $t_1$ and $t_2$
associated with different exit points from state $n_1$]

The change steps that are used to evolve from Milestone 3 to Milestone 4 are
given in Table 3.4. Milestone 4 introduces a mapping from exit points to transition
targets (4.2,4.3). The mapping is applied to all exit points in the state machine and
the transitions that originate from those exit points. The target of these transitions
resolves to either a state process instantiation or, in the case of a continuation to the
next higher level in the state hierarchy, another exit point (4.1).

### 3.3.5 Milestone 5: group transitions

Group transitions represent another case of transition configurations. In the group
transition case, a transition is an “incoming” transition and also the first transition
in a chain. This requires that composite states handle possible transitions, whereas
### 3.3. TRANSFORMATION SPECIFICATION MILESTONES

<table>
<thead>
<tr>
<th>Id</th>
<th>Description</th>
<th>M3 Context</th>
<th>M4 Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Define “outgoing transition target” mapping</td>
<td>n/a</td>
<td>1.12: ( Q_i := \ldots )</td>
</tr>
<tr>
<td>4.2</td>
<td>Add “exit point” process definition</td>
<td>n/a</td>
<td>1.17: ( B_i := \ldots )</td>
</tr>
<tr>
<td>4.3</td>
<td>Insert “exit point” process definition in “composite state” local definition</td>
<td>l.17: ( \ldots \text{def } { D_1; \ldots; D_k; C_{\text{def}} } )</td>
<td>1.14: ( \ldots \text{def } { D_1; \ldots; D_k; B_k; \ldots; B_m; C_{\text{def}} } )</td>
</tr>
<tr>
<td>4.4</td>
<td>Replace “target state” process instantiation with “transition target” process instantiation in basic state event handler</td>
<td>1.2: ( \ldots \text{x}<em>i.S</em>{n_i}(p_i) )</td>
<td>1.2: ( \ldots \text{x}_i.Q_i )</td>
</tr>
</tbody>
</table>

Table 3.4: Change steps from Milestone 3 to Milestone 4

Preceding milestones assume that only basic states need to handle transitions. In the example shown in Figure 3.14, the state machine will initialize into state n2. From state n2, transition t1 can be triggered, causing the state machine to first exit state n2, then exit state n1, execute transition t1, and finally enter state n3.

![Figure 3.14: UML-RT state machine example for Milestone 5: t1 is a group transition](image)

The change steps that are used to evolve from Milestone 4 to Milestone 5 are given in Table 3.5. Milestone 5 introduces event handlers in the composite state processes (5.9, 5.10, 5.11) to capture events that trigger the group transitions. Further, exit events are passed as parameters to each state (5.1, 5.4), so that when a group transition is taken, the child states processes will terminate in the correct order (5.2).
### 3.3. TRANSFORMATION SPECIFICATION MILESTONES

<table>
<thead>
<tr>
<th>Id</th>
<th>Description</th>
<th>M4 Context</th>
<th>M5 Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Add exit handler parameters to basic state process definition</td>
<td>1.2: ... proc $S_n(\text{enp})$ ...</td>
<td>1.2: ... proc $S_n(\text{enp, exit, exack})$ ...</td>
</tr>
<tr>
<td>5.2</td>
<td>Add “exit” listen branch to basic state event handler</td>
<td>1.2: ... $x_i ? Q_i$</td>
<td>1.2: ... $x_i ? Q_i + \text{exit?}.exack$!</td>
</tr>
<tr>
<td>5.3</td>
<td>Add exit handler parameters to state process instantiation in mapping for “$Q_i$”</td>
<td>1.12: $S_n_i(p_i)$ ...</td>
<td>1.12: $S_n_i(\text{exit, exack, sh})$ ...</td>
</tr>
<tr>
<td>5.4</td>
<td>Add exit handler parameters to composite state process definition</td>
<td>1.14: ... proc $S_n(\text{enp})$ ...</td>
<td>1.15: ... proc $S_n(\text{enp, exit, exack})$ ...</td>
</tr>
<tr>
<td>5.5</td>
<td>Add termination event trigger to exit point process definition “$B_q_i$”</td>
<td>1.17: ... $= Q_j$</td>
<td>1.22: ... $(\text{sh}!!</td>
</tr>
<tr>
<td>5.6</td>
<td>Add exit parameters to entry dispatcher process definition “$C$”</td>
<td>1.20: proc $C(\text{enp}) = $</td>
<td>1.25: proc $C(\text{enp, exit, exack, sh}) = $</td>
</tr>
<tr>
<td>5.7</td>
<td>Add inner-state exit parameters to target state process instantiation in entry dispatcher process “$C$”</td>
<td>1.21-24: $S_{n_1}(p'<em>1)$ ... $S</em>{n_2}(p'<em>2)$ ... $S</em>{n_m}(p'_m)$</td>
<td>1.26-29: $S_{n_1}(p'<em>1, \text{exit}', \text{exack}', \text{sh}')$ ... $S</em>{n_m}(p'_m, \text{exit}', \text{exack}', \text{sh}')$</td>
</tr>
<tr>
<td>5.8</td>
<td>Add inner-state exit parameters to default state process instantiation in entry dispatcher process “$C$”</td>
<td>1.25: else $S_{n_d}(p'_d)$</td>
<td>1.30: else $S_{n_d}(p'_d, \text{exit}', \text{exack}', \text{sh}')$.</td>
</tr>
<tr>
<td>5.9</td>
<td>Add process definition “$H$” for handling events in composite states</td>
<td>n/a</td>
<td>1.36-38: proc $H(\text{exit}', \text{exack}', \text{sh}') = $ ...</td>
</tr>
<tr>
<td>5.10</td>
<td>Insert composite event handler process definition “$H$” in composite state local definition</td>
<td>1.14: def ${\ldots; C_{def}}$</td>
<td>1.16: def ${\ldots; C_{def}; H_{def}}$</td>
</tr>
<tr>
<td>5.11</td>
<td>Replace composite state local definition process with one that executes updated “$C$” and “$H$” process instances in parallel</td>
<td>1.14: ... in $C(\text{enp})$</td>
<td>1.17-19: new $\text{exit}', \text{exack}', \text{sh}'$ in ... $H(\text{exit}', \text{exack}', \text{sh}')$.</td>
</tr>
</tbody>
</table>

Table 3.5: Change steps from Milestone 4 to Milestone 5
Chapter 4

Data Gathering Approach

Figure 4.1 describes the data gathering process. It is important to emphasize that the data gathered for this work represents an exploratory demonstration and not a full empirical experiment. This process simply serves to demonstrate how the metrics are obtained and why the specific set of metrics were chosen. It can be re-purposed to perform a more comprehensive empirical experiment.

![Figure 4.1: The data gathering process. Versions of transformation implementation artifacts are converted to an abstract representation. Difference models are calculated from adjacent versions.](image-url)
4.1 Procedure

The creation of data for this demonstration involves implementing the transformation described in Chapter 3. First, the transformation task’s specification is broken down into milestones. Each milestone describes a valid mapping definition from UML-RT state machines to Kiltera process definitions.

The change steps defined within each milestone in Chapter 3 are used to guide the implementation of the transformation. For each milestone, steps are implemented in succession in the language to be evaluated. Changes are committed whenever the implementation artifact returns to a “valid” state, according to the language tool. For example, in ATL, whenever the editor’s syntax checker reports no errors after a change, then it can be committed to the version control repository. In RSA, when the mapping editor reports no errors and there are no Java syntax errors, the transformation artifacts can be committed. Although this method of committing changes requires more effort during implementation, it provides flexibility when analysing the change data since the change granularity can be considered at different scales. When the change steps are complete, the transformation implementation is refactored with the goal of simplification and minimization. This signals the end of one milestone of development.

Finally, difference models are extracted from adjacent commits of the implementation artifacts. The EMF Compare tool is used to create these difference models. The inputs provided to EMF Compare are transformation artifacts, converted into an EMF format.
4.1. PROCEDURE

4.1.1 Version Control Repository

Subversion, a revision control system, is used to track changes made to implementations. Each commit is exported from the version control system and then translated into an abstract representation format suitable for comparison. The implementation artifacts are converted to an EMF-based\footnote{See http://www.eclipse.org/emf} XMI format if not already represented as such. Extraction of the commit revisions is done programmatically, leveraging the API of the Eclipse Subversive tool. The extraction process is integrated into Eclipse as a context menu action (refer to Section 4.2 for details).

4.1.2 Language Representation

RSA Mapping Artifacts

RSA mapping files are serialized as XML, however there is an Eclipse plugin that loads the XML data into an EMF model. This EMF model can then be serialized back out as an EMF-based XMI file. In the original RSA Mapping artifacts, custom Java code can exist both internally in the XML mapping file, or externally as full Java class files. In both cases, the Java code is serialized as plain text. The text-based Java code must be converted into a model form and then integrated with the RSA Mapping model.

This modelization of Java is accomplished using JAMOPP [25]. JAMOPP is a project that provides a representation of the Java abstract syntax tree as an EMF metamodel. The JAMOPP project also includes a parser and an unparsable which facilitates conversion between the concrete textual syntax and the abstract model-based syntax. The JAMOPP runtime parses Java code fragments into corresponding
EMF-based model representations. These models of the code are then inserted in place of the original Java text-based code in the RSA Mapping model.

In the original RSA Mapping metamodel, instances of the CodeRefinement class represent the Java customizations associated with each mapping rule. To support the embedding of JAMOPP models in RSA Mapping models, the RSA Mapping metamodel is extended to subclass CodeRefinement with a new JavaModelCodeRefinement that contains a JAMOPP Method reference. The RSA Mapping artifacts are converted to the extended metamodel and the original CodeRefinement instances are replaced.

**ATL Artifacts**

ATL files are serialized as plain text source files, but the parser that comes with the ATL development tools can generate EMF models from these source files. This functionality provided by the ATL parser is used to generate input artifacts for the differencing step of the data gathering process. However, the default configuration of the parser inserts trace locations for every language token in the ATL file. Trace locations consist of line and column numbers in the original textual syntax that map to each element. Any change in an ATL source file will affect the location of every following token in the file. The result from EMF Compare will typically contain a large number of extraneous changes that interfere with change metric calculations. To avoid this issue, the ATL parser is configured to disable the insertion of trace locations.
4.1.3 Model Differencing

Difference models are extracted using a tool called EMF Compare\textsuperscript{2}. The tool accepts input models in XMI format and it outputs EMF models typed over an Ecore metamodel that is specific to EMF Compare. Only artifacts of the same language in successive commits are differenced, not artifacts of different languages. Reasons for selecting the EMF Compare tool are:

- built-in support for Ecore and XMI input formats
- support for similarity-based differencing
- a model-based representation of difference results
- extensibility features using the included plugin extensions and API

The architecture of EMF Compare is shown in Figure 4.2. EMF Compare uses a two-step process to calculate the differences. The process is based on the work by Xing et al. [54]. First, the MatchEngine component builds a match model. The match model represents all detected mappings between similar elements from the files being compared. Once the match model is created, it is used to calculate differences. The resulting difference model drives a user interface that displays the differences graphically. However the difference model can also be serialized and manipulated using modeling APIs and tools. This serialized model is used to gather change metrics from adjacent versions of the transformation artifacts.

Even though EMF Compare provides similarity-based model differencing, an initial evaluation of the tool on the outliers of comparisons from Milestone 1 revealed

\textsuperscript{2}See http://www.eclipse.org/emf/compare
that some modifications to its similarity heuristics were necessary. There were two modifications made to EMF Compare in order to better support the changes found in the data; an adjustment of the MatchEngine heuristics and an extension of the DiffEngine to support a new type of change.

**MatchEngine heuristics**

![Figure 4.3: An example model modification: adding a Helper to the Module root. Two separate MatchEngine behaviours are shown. Matches are denoted by dashed lines. Detected changes are denoted by labels beside the elements.](image)

The difficulty in relying on heuristics in the match phase shows when the heuristic rules and weights are poorly fitted to the data. This may cause spurious errors which
are hard to detect. For example, inserting a new element into an existing list of elements may additionally trigger pairs of removal and insertion changes for the elements that follow. Figure 4.3 shows both the default behaviour of the MatchEngine as well as the behaviour that is desired. In the default behaviour, the match result causes EMF Compare to detect extraneous changes for the deletion of the original Rule1, the modification of Rule2 into Rule1, and a re-addition of the original Rule2. The modified behaviour’s match result causes EMF Compare to detect only the addition of the Helper element, as desired. To understand why the default behaviour occurs, it is necessary to consider how the statistics are calculated and how they are weighted.

In a two-way diff, the default MatchEngine uses an aggregate difference metric to determine whether two elements (one from each EMF Compare input) form a match. The aggregate metric is comprised of a weighted sum of statistical metrics which is then normalised. Part of the contribution to the aggregate metric is a structure similarity measure. This measure is based on the containment and relative position of elements in the models. In the default MatchEngine, half of the weight of the structure similarity measure is given to a boolean comparison of the URI fragments of the two elements, in other words whether the paths from the root of the model are equal.

When a new element is inserted in the model, as shown in Figure 4.3, the URI fragments of the existing elements following it will change. URI fragment similarity is weighted high enough that it can prevent correct matches for an element even though none of the contents or features of the element are different. In order to mitigate this behaviour, the weight of the URI fragment comparison test is decreased and the weight of the NameSimilarity measure, which is essentially the string distance of the
element name, is increased. The prioritization of name similarity over URI fragment equality allows the MatchEngine to find the correct matches in this case. However, even though this rebalancing of heuristic weights may help for the types of models encountered in the transformation task, it is not a general solution.

**Type generalization/specialization change**

Element type changes are another category of change that is not addressed by the default implementation of EMF Compare. This kind of change arises when a model element’s type class is either changed to a specialization of the original type, or a generalization of the original type. For example, in ATL, sometimes a regular MatchedRule element was changed to a LazyMatchedRule element. In terms of concrete textual syntax, this involved simply prepending the keyword 'lazy' to the rule declaration. In the abstract syntax, the MatchedRule class is a supertype of the LazyMatchedRule class so this is a specializing type change. In fact, their effective features are identical and the only differences between the two are their class names and their behavioural semantics.

By default, EMF Compare sees this change as a removal of the subgraph for the MatchedRule element and an addition of a subgraph for the new LazyMatchedRule element with identical descendants. Figure 4.4(a) shows how the default EMF Compare MatchEngine and DiffEngine handles this change, and 4.4(b) shows the proposed EMF Compare behaviour. In the proposed behaviour, there is only a single change that represents a change of type of the element.

There is no class in the DiffElement hierarchy of EMF Compare’s metamodel that specifically captures this type of change. In the interest of saving time and
4.2 DATA GATHERING TOOLS

![Diagram](image)

(a) default EMF Compare behaviour

(b) modified EMF Compare behaviour

Figure 4.4: Example model modification: changing the type of an element from MatchedRule to LazyMatchedRule. Matches are denoted by dashed lines. Detected changes are denoted by labels beside the elements.

modification effort, an existing class in the DiffElement hierarchy, ElementChange, is repurposed to represent the new type of change. The use of this class should not conflict with the existing DiffEngine implementation since it is not originally a representation of a concrete change type. The default EMF Compare DiffEngine is extended to create an ElementChange instance whenever it detects a type change. Finally, the metric calculation process is modified to handle ElementChange instances.

4.2 Data Gathering Tools

The prototype tools developed to facilitate data gathering for this thesis work are integrated into the Eclipse framework as plugins. There are three categories of functions provided by the tool plugins: revision extraction, calculation of metrics, and the
user interface/extraction workflow. The user interface is decoupled from the tool’s operations logic so that it can be easily modified if the requirements of the extraction process change.

Figure 4.5 shows the workflow for extracting versioned artifacts from the repository. First, the user selects the versioning-enabled resource from the File Navigator view. Right-clicking on the resource reveals a context menu item that opens the SVN Export dialog. In this dialog, the user can select an output folder where exported versions will be stored. The dialog also provides a text box to specify a list of version ranges. These ranges are then used to limit the export. Finally, a list of SVN property names can be provided to extract SVN metadata about the versions. This data will be stored in a separate text file.

4.2.1 User Interface

![Image of SVN Export context menu item](image1)

![Image of SVN Export dialog](image2)

Figure 4.5: The workflow for exporting repository artifacts
4.3. CHARACTERISING TRANSFORMATION ARTIFACTS

The user-interface for calculating metrics from versions of the transformation artifacts is the same for both model-change metrics and language-specific metrics. A set of transformation artifact resources can be selected in the Eclipse Project Navigator. Depending on which resources are selected, the appropriate action will be shown in the right-click context menu. Upon launching the tool, the user selects an output folder from a dialog, as shown in Figure 4.6. The metrics calculated from the selected resources are then placed in a comma-separated values (CSV) file in the output folder.

![Figure 4.6: Change metrics output selection dialog.](image)

### 4.3 Characterising Transformation Artifacts

Transformation artifacts can be characterised by metrics and qualitative properties. We consider two types of metrics: static language metrics such as those proposed in [2] and metrics based on the change between versions of the transformation implementation. Static language metrics, for example the number of mapping rules or rule dependency measures, provide a profile of the transformation implementation at
each commit version. Change metrics provide a direct indicator for the activity that occurs between commit versions. The number of elements added in a commit is an example of a change metric.

### 4.3.1 Transformation Language Metrics

We have implemented static metrics for ATL based on the metrics presented in [2], and we have defined comparable metrics for the mapping tool provided by the IBM RSA modeling environment. For each language, metrics are chosen to represent one of four properties: size, inheritance complexity, and dependency. A summary of the static metrics used in this paper is shown in Table 4.1. Metrics that are shown adjacent between the two languages measure similar elements in their respective languages.

![Table 4.1: Static metrics for RSA and ATL artifacts.](image)

<table>
<thead>
<tr>
<th>Category</th>
<th>ATL Metric</th>
<th>RSA Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td># rules</td>
<td># mapping declarations</td>
</tr>
<tr>
<td></td>
<td># helpers</td>
<td></td>
</tr>
<tr>
<td></td>
<td># pattern elements/rule</td>
<td></td>
</tr>
<tr>
<td></td>
<td># rule bindings/rule</td>
<td></td>
</tr>
<tr>
<td>Inheritance</td>
<td># abstract rules</td>
<td></td>
</tr>
<tr>
<td>Complexity</td>
<td># collection operations/rule</td>
<td># loops/mapping declaration</td>
</tr>
<tr>
<td></td>
<td>cyclomatic complexity/rule</td>
<td>cyclomatic complexity/mapping decl.</td>
</tr>
<tr>
<td>Dependency</td>
<td>lazy rule fan-out</td>
<td>submap fan-out</td>
</tr>
<tr>
<td></td>
<td>lazy rule fan-in</td>
<td>mapping definition fan-in</td>
</tr>
<tr>
<td></td>
<td>helper fan-out</td>
<td></td>
</tr>
<tr>
<td></td>
<td>helper fan-in</td>
<td></td>
</tr>
</tbody>
</table>
4.3. CHARACTERISING TRANSFORMATION ARTIFACTS

Definitions of ATL Metrics

Each metric value is calculated for all the transformation artifacts of a specific version of the transformation implementation. Where appropriate, a set of minimum, maximum, and mean values are calculated. For example, the number of pattern elements per rule is calculated as the minimum number of pattern elements out of all rules, the maximum out of all rules, and the mean number of pattern elements for all rules.

1. **Number of Rules**: total count of matched, lazy, and called rules

2. **Number of Helpers**: count of helpers

3. **Number of pattern elements per rule**: count of in-pattern and out-pattern elements in each specific rule

4. **Number of rule bindings per rule**: count of rule bindings in all out-pattern elements in each specific rule

5. **Number of abstract rules**: count of abstract matched rules

6. **Number of collection operations per rule**: count of all CollectionOpExp elements and for loop expression elements in each specific rule

7. **Cyclomatic complexity per rule**: based on McCabe’s definition of cyclomatic complexity [41], counting only conditionals and logical expressions in each specific rule

8. **Lazy rule fan-out**: count of lazy rule references in each specific rule

9. **Lazy rule fan-in**: count of all references to each specific lazy rule
4.3. CHARACTERISING TRANSFORMATION ARTIFACTS

10. **Helper fan-out**: count of helper references in each specific rule

11. **Helper fan-in**: count of all references to each specific helper

Given the nature of the transformation implemented here, many of the metrics listed in [2] are not applicable. For example, there are no metrics that involve called rules because the ATL transformation artifacts in the implemented transformation do not contain any called rules, and very few lazy rules. There are no operation helpers and the helpers do not have any parameters. Granted, in general a transformation can evolve to use additional language features. However the transformation task that we consider in this thesis is closed so these unused metrics are left out of analysis for simplicity.

The definition of the fan-out metric for ATL is not identical to any of the dependency metrics described in [2]; instead it specifically measures lazy rules. The reason for this new metric is because ATL matched rule dependency can’t be measured directly with simple static analysis of the OCL expressions contained in ATL rules. Originally, a metric that included navigation expressions was devised in an attempt to measure fan-out of all types of rules and helpers. However, it was found that this metric was not effective in measuring fan-out. Section 5.1 contains details of the analysis and also discusses challenges in finding relevant metrics.

**Definitions of RSA Metrics**

As with the set of ATL metrics, each metric value is calculated for all the transformation artifacts of a specific version of the transformation implementation. Where appropriate, a set of minimum, maximum, and mean values are calculated.
4.3. CHARACTERISING TRANSFORMATION ARTIFACTS

1. **Number of mapping declarations**: count of all mapping declaration elements

2. **Number of refining mappings per mapping declaration**: count of refining mapping elements in each specific mapping declaration

3. **Number of loops per mapping declaration**: count of all for loop statement elements in each specific mapping declaration

4. **Cyclomatic complexity per mapping declaration**: based on McCabe’s definition of cyclomatic complexity [41], counting only conditionals and logical expressions in each specific mapping declaration

5. **Submap fan-out**: count of all submap and custom submap refinements in each specific mapping declaration.

6. **Mapping declaration fan-in**: count of references to each specific mapping declaration, from all submap and custom submap refinements

The RSA mapping framework’s tool does not provide as many language features as ATL, so the mapping models tend to be more simple than those in ATL. This is reflected in the smaller number of language metrics that are relevant for the transformation task.

4.3.2 Model Change Metrics

Model change metrics are calculated from the difference models. Four different types of changes are considered: addition, deletion, update, and move. A count of these
Table 4.2: Basic change operations and where they apply.

<table>
<thead>
<tr>
<th>Change Operation</th>
<th>Applied To</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>Elements,</td>
</tr>
<tr>
<td>Add</td>
<td>Multi-value properties (references and attributes)</td>
</tr>
<tr>
<td>Delete</td>
<td>Elements</td>
</tr>
<tr>
<td>Delete</td>
<td>Multi-value properties (references and attributes)</td>
</tr>
<tr>
<td>Move</td>
<td>Elements</td>
</tr>
<tr>
<td>Update</td>
<td>Single-value properties (references and attributes)</td>
</tr>
<tr>
<td>Update Class</td>
<td>Elements</td>
</tr>
</tbody>
</table>

changes provides an estimate of the level of activity associated with a task or milestone (or any group of commits).

Table 4.2 summarizes the change operations and how each type of change applies to various parts of the language artifact model. EMF Compare’s difference metamodel and the structure of EMF-based models are leveraged to calculate the change count metrics. Addition and deletion changes can apply to model elements, multi-valued attributes and multi-valued references. Move changes can only be applied to model elements. Update changes can be made to single-valued attributes and single-valued references. Additionally, an element’s type may be changed; the resulting type must either be a specialization of the current type or a generalization of the current type. An element type change is considered as an update change.

Another class of change metrics describes the characteristics of individual changes. These are **descriptive change metrics** because the metrics are derived from the size, shape, and composition of individual changes. A summary of these changes are shown in Table 4.3. These metrics take advantage of the graph-based nature of
4.3. CHARACTERISING TRANSFORMATION ARTIFACTS

Table 4.3: Descriptive change metrics that characterize individual changes.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Derived From</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>Element Changes</td>
<td>$1 + \text{number of contained elements}$</td>
</tr>
<tr>
<td></td>
<td>Property, Update Class Changes</td>
<td>1</td>
</tr>
<tr>
<td>Shape (height)</td>
<td>Element Changes</td>
<td>maximum depth of subgraph</td>
</tr>
<tr>
<td></td>
<td>Property, Update Class Changes</td>
<td>1</td>
</tr>
<tr>
<td>Shape (breadth)</td>
<td>Element Changes</td>
<td>maximum breadth of subgraph</td>
</tr>
<tr>
<td></td>
<td>Property, Update Class Changes</td>
<td>1</td>
</tr>
<tr>
<td>Depth</td>
<td>All changes</td>
<td>depth of the changed element</td>
</tr>
<tr>
<td>Composition</td>
<td>Element changes</td>
<td>number of class types in subgraph</td>
</tr>
<tr>
<td></td>
<td>Property, Update Class Changes</td>
<td>1</td>
</tr>
</tbody>
</table>

Model artifacts. The size of an element change can be represented by the number of descendant elements from the root, inclusive. Changes to properties and update changes are considered to have a size of 1. The shape of element changes can be represented by the maximum height of the element’s subgraph as well as the maximum breadth. Breadth is calculated by counting the directly contained elements of each element in the subgraph and taking the maximum. The depth metric is an indication of where the change occurs in the tree. It is calculated as the depth of the change’s root element in the model artifact graph. Finally, the composition of changes is represented by the number of unique class types present in the element’s subgraph. The metrics that are calculated using the root element of the subgraph are inclusive of the root element.
4.4 Summary of Collected Data

Table 4.4: Artifacts collected for the demonstrated transformation task.

<table>
<thead>
<tr>
<th>Artifact Type</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ATL</td>
</tr>
<tr>
<td>Milestones implemented</td>
<td>5</td>
</tr>
<tr>
<td>Commits</td>
<td>223</td>
</tr>
<tr>
<td>Changes</td>
<td>665</td>
</tr>
</tbody>
</table>

Through the described data gathering process, data sets are collected for each language as shown in Table 4.4. The cumulative change size is the sum of the number of changes of all commits.

Figure 4.7 indicates the overall distribution of commits across milestones M1 to M4 for implementations using ATL and the RSA mapping tool. Comparing the number of commits required for each milestone, ATL and RSA implementations for M1 and M3 are similar. However, the ATL implementation requires more commits to complete M2 and M4, and there are more commits in the ATL implementation compared to the RSA mapping implementation.

The commits can also be mapped to specification change steps. This reveals some notable relationships between the specification and the implementation. For example, occasionally two change steps are addressed by the same commit such as steps 5.7 and 5.8 in the ATL implementation and steps 3.3 and 3.5 in the RSA implementation. This suggests that change steps may be coupled depending on the features of the implementation language and the language tools, and not just within
Another observation is that the commits for a given change step are not necessarily contiguous. There are instances where an incomplete implementation of the change step is not noticed right away, or a bug can only be caught after the next runnable version of the implementation is complete. Several corrections were applied to change step 2.3, even as late as the implementation of Milestone 5. The change step involves a relatively complicated OCL Helper expression.

Finally the implementation of change steps are sometimes ordered differently, due to the workflow of the language tool, or the influence of language features. Consider
a transformation implementation at change step 4.1. In the ATL transformation, change step 4.3 was chosen next for implementation, whereas in the RSA mapping transformation, change step 4.2 was chosen. In the RSA implementation, often the mapping declarations are created “on-the-fly” when the submap refinement mapping is created using the tool. This tends to encourage a top-down approach when building the transformation. In ATL new rules and their corresponding references are specified separately and the order is not influenced as much by the tool.
Chapter 5

Analysis

This chapter presents in detail how the data gathered from the transformation task can be used to perform exploratory analysis. The first section of this chapter is related to the Data Understanding and Data Preparation parts of the data mining process. This includes statistical characterization of the data and an evaluation of the ATL fan-out metric that was proposed in the Chapter 4. The second section of the analysis describes an exploratory analysis using unsupervised clustering to see if any structure can be found in the observations. Finally, the third section describes the creation of a classifier to detect types of software evolution. All examples are performed using the software environment R v2.15.1.

5.1 Evaluation of Fan-out Metric

This section provides an example of evaluating a metric, showing some of the issues that arise when attempting to define metrics. When determining the language metrics that would be collected for the ATL language artifacts in Chapter 4, a metric was devised to capture the overall rule and helper fan-out of the transformation rules. Unfortunately the metric was found to not be representative of overall fan-out. One
way to verify this is to perform a Spearman rank correlation test on the fan-out metric and the actual number of times a rule or helper is invoked. The Spearman test is selected because it is non-parametric and because the sample size is small. Since the OCL expressions in the ATL artifacts tend to accumulate with increasing commits and rarely change, we consider the last committed ATL source artifact of Milestone 4. There is a significant correlation between the two samples for $\alpha = 0.01 (\rho = .73, df = 10, P = .007)$.

However, the proposed metric is actually a sum of two metrics: call expressions to lazy-rules, and navigation expressions. A second Spearman rank correlation test shows that after separating the two parts, the navigation expression metric is found to not correlate significantly with a fan-out dependency on matched rules and helpers ($\rho = .17, df = 10, P = .96$). The lazy-rule call metric is by definition a direct measure, so it is used instead of the originally proposed metric for fan-out dependency.

The observation illustrates the sensitivity of the analysis to feature selection. In this demonstration, it can be challenging to find metrics that:

1. apply to the artifacts being analysed, and

2. are related to the properties that are being analysed through prior evidence or intuition.

For example, the ATL transformation does not make use of many advanced language features and it is fully contained within a single file throughout its entire evolution. This negates the usefulness of metrics that reflect multiple transformation units and libraries. There are no called rules and so any metrics related to called rules are not of any use in this demonstration. Neither are metrics related to other advanced features. Similar issues are encountered for the RSA mapping artifacts.
5.2 Exploratory Statistical Analysis

A useful step in characterizing the nature of the data statistically is to determine a suitable model for the probability distribution of the attributes in the data set. This information is important because it influences the selection of algorithms and methods in further analysis. We use the Shapiro-Wilk test to first determine whether each attribute’s distribution is non-normal (the null hypothesis of the test is that a sample is a representation of a normal distribution). The reason for selecting the Shapiro-Wilk test is that it is more appropriate for small sample sizes.

At the individual commit level, all attributes, both change metrics and language metrics, exhibit significant non-normality for a chosen $\alpha = 0.05$. At the change step level of granularity for the ATL data, two of the change attributes avoid rejection of the null hypothesis: average depth and maximum depth. The RSA change step data also tends toward normality; average size, average width, average diversity, and the number of mapping declarations avoid rejection. Figure 5.1 illustrates the normality tendency using histogram plots. The change step observations, being aggregates of the commit observations, exhibit less skew.

However, for the chosen level of significance, most of attributes remain non-normal at the change step level. The prevalence of non-normal attributes as well as the small sample sizes (44 samples at the change step level of granularity for ATL and 20 samples for RSA), suggest that non-parametric techniques may be more suitable for analysis.

We can also examine the correlation between each attribute using a correlation matrix. For aggregate metrics with minimum, maximum, and average values, only the correlation of the average metrics are included. Only matrices for the change step
5.2. EXPLORATORY STATISTICAL ANALYSIS

![Histogram and estimated density plot for depth_avg at commit granularity and at change step granularity]

Figure 5.1: Histogram and estimated density plot for depth_avg at commit granularity and at change step granularity

data sets are presented, however differences with the commit granularity data sets are discussed. The Pearson correlation coefficient is chosen over the Spearman correlation coefficient because the Spearman coefficient looks for monoticity in the relationship between two variables. This may not be an accurate indicator for attributes that do not change often. Table 5.1 lists some abbreviations for the metrics, which will be used for the rest of the analysis.

Tables 5.2 and 5.3 contain correlation coefficients between attributes generated from the ATL change metrics dataset and the RSA mapping change metrics dataset, respectively. The numeric value is the Pearson correlation coefficient, and the stars represent the p-value significance level. One star indicates that $p < 0.05$, two stars indicate that $p < 0.01$, and three stars indicate that $p < 0.001$.

The change metric correlation matrices show that many of the descriptive change metrics correlate significantly with the total size of the changes and with each other. A similar effect is described in literature on software measurement, where many metrics relating to language, change dependency and complexity are found to be correlated to
5.2. EXPLORATORY STATISTICAL ANALYSIS

Table 5.1: Metrics abbreviations

<table>
<thead>
<tr>
<th>Category</th>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>common</td>
<td>cmt</td>
<td>commit identifier</td>
</tr>
<tr>
<td></td>
<td>lbl</td>
<td>changestep label</td>
</tr>
<tr>
<td></td>
<td>mn</td>
<td>minimum</td>
</tr>
<tr>
<td></td>
<td>mx</td>
<td>maximum</td>
</tr>
<tr>
<td></td>
<td>av</td>
<td>average</td>
</tr>
<tr>
<td>change count metrics</td>
<td>crt</td>
<td>creates</td>
</tr>
<tr>
<td></td>
<td>upd</td>
<td>updates</td>
</tr>
<tr>
<td></td>
<td>del</td>
<td>deletes</td>
</tr>
<tr>
<td></td>
<td>mov</td>
<td>moves</td>
</tr>
<tr>
<td></td>
<td>tot</td>
<td>total</td>
</tr>
<tr>
<td>descriptive change metrics sz</td>
<td>sz</td>
<td>size</td>
</tr>
<tr>
<td></td>
<td>hgt</td>
<td>height</td>
</tr>
<tr>
<td></td>
<td>wdt</td>
<td>width</td>
</tr>
<tr>
<td></td>
<td>dpt</td>
<td>depth</td>
</tr>
<tr>
<td></td>
<td>div</td>
<td>diversity</td>
</tr>
<tr>
<td>ATL language metrics</td>
<td>nrl</td>
<td>number of rules</td>
</tr>
<tr>
<td></td>
<td>nhp</td>
<td>number of helpers</td>
</tr>
<tr>
<td></td>
<td>nar</td>
<td>number of abstract rules</td>
</tr>
<tr>
<td></td>
<td>pelem</td>
<td>pattern elements per rule</td>
</tr>
<tr>
<td></td>
<td>rbnd</td>
<td>rule bindings</td>
</tr>
<tr>
<td></td>
<td>col</td>
<td>collection operations</td>
</tr>
<tr>
<td></td>
<td>cmp</td>
<td>cyclomatic complexity</td>
</tr>
<tr>
<td></td>
<td>lrfo</td>
<td>lazy rule fan-out</td>
</tr>
<tr>
<td></td>
<td>lrfi</td>
<td>lazy rule fan-in</td>
</tr>
<tr>
<td></td>
<td>hpfo</td>
<td>helper fan-out</td>
</tr>
<tr>
<td></td>
<td>hpli</td>
<td>helper fan-in</td>
</tr>
<tr>
<td>RSA mapping language metrics</td>
<td>nmpd</td>
<td>number of mapping declarations</td>
</tr>
<tr>
<td></td>
<td>mp</td>
<td>mappings per mapping declaration</td>
</tr>
<tr>
<td></td>
<td>mpdfo</td>
<td>submap refinement fan-out</td>
</tr>
<tr>
<td></td>
<td>mpdfi</td>
<td>mapping declaration fan-in</td>
</tr>
<tr>
<td></td>
<td>cmp</td>
<td>cyclomatic complexity</td>
</tr>
<tr>
<td></td>
<td>col</td>
<td>for loops</td>
</tr>
</tbody>
</table>

the number of lines of code [19]. Additionally, the "create" and "delete" change types correlate significantly with the descriptive metrics, likely because these operations change the number of elements in the transformation artifact, thereby affecting the size of the changes.

The correlation coefficients between language metrics in the ATL dataset and the RSA dataset, are presented in Tables 5.4 and 5.5, respectively. It appears that most of the attributes are strongly correlated with each other. Again, this makes sense since it is known that many types of metrics correlate in terms of size.
5.2. EXPLORATORY STATISTICAL ANALYSIS

Table 5.2: Pearson correlation coefficients between ATL change metrics.

<table>
<thead>
<tr>
<th></th>
<th>crt</th>
<th>upd</th>
<th>del</th>
<th>mov</th>
<th>tot</th>
<th>sz</th>
<th>hgt</th>
<th>wdt</th>
<th>dpt</th>
</tr>
</thead>
<tbody>
<tr>
<td>crt</td>
<td>0.94***</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>upd</td>
<td></td>
<td>0.89***</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>del</td>
<td></td>
<td></td>
<td>0.11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tot</td>
<td>0.97***</td>
<td>1.00***</td>
<td></td>
<td>0.95***</td>
<td></td>
<td>0.05</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sz</td>
<td>0.13</td>
<td>0.16</td>
<td>0.20</td>
<td></td>
<td>-0.07</td>
<td>0.16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hgt</td>
<td>-0.08</td>
<td>-0.03</td>
<td>0.01</td>
<td>-0.05</td>
<td>-0.04</td>
<td>0.89***</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wdt</td>
<td>0.03</td>
<td>0.01</td>
<td>0.07</td>
<td>-0.08</td>
<td>0.02</td>
<td>0.80***</td>
<td>0.60***</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dpt</td>
<td>-0.01</td>
<td>0.05</td>
<td>0.07</td>
<td>0.08</td>
<td>0.04</td>
<td>-0.33*</td>
<td>-0.45**</td>
<td>-0.16</td>
<td></td>
</tr>
<tr>
<td>div</td>
<td>0.00</td>
<td>0.00</td>
<td>0.04</td>
<td>-0.05</td>
<td>0.00</td>
<td>0.91***</td>
<td>0.87***</td>
<td>0.74***</td>
<td>-0.51***</td>
</tr>
</tbody>
</table>

Table 5.3: Pearson correlation coefficients between RSA change metrics.

<table>
<thead>
<tr>
<th></th>
<th>crt</th>
<th>upd</th>
<th>del</th>
<th>mov</th>
<th>tot</th>
<th>sz</th>
<th>hgt</th>
<th>wdt</th>
<th>dpt</th>
</tr>
</thead>
<tbody>
<tr>
<td>crt</td>
<td>0.41</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>upd</td>
<td></td>
<td>0.53*</td>
<td>0.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>del</td>
<td></td>
<td></td>
<td>0.54*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tot</td>
<td>0.88***</td>
<td>0.75***</td>
<td></td>
<td>0.85***</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sz</td>
<td>0.13</td>
<td>-0.11</td>
<td>-0.22</td>
<td>-0.04</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hgt</td>
<td>0.26</td>
<td>0.08</td>
<td>0.01</td>
<td>0.20</td>
<td>0.85***</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wdt</td>
<td>-0.22</td>
<td>-0.56*</td>
<td>-0.38</td>
<td>-0.50*</td>
<td>0.52*</td>
<td>0.99</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dpt</td>
<td>0.09</td>
<td>0.71***</td>
<td>0.04</td>
<td>0.42</td>
<td>-0.31</td>
<td>-0.04</td>
<td>-0.74***</td>
<td></td>
<td></td>
</tr>
<tr>
<td>div</td>
<td>0.11</td>
<td>-0.13</td>
<td>-0.25</td>
<td>-0.07</td>
<td>0.98***</td>
<td>0.85***</td>
<td>0.50*</td>
<td>-0.35</td>
<td></td>
</tr>
</tbody>
</table>

Difference in changes per commit between ATL and RSA mapping datasets

The medians of the number of changes per commit are presented in Figure 4.7. In this section we determine whether the observations for the ATL dataset and the observations for the RSA mapping dataset tend to be different with any significance. A Mann-Whitney U test can be performed to determine this information. It is an unpaired non-parametric test for ordinal, independent variables. Given two sample populations X and Y, the null hypothesis is that the probability of a sample from X being larger than a sample from Y is equal to the probability of a sample from Y being larger than a sample of X. Section 4.4 reports that the median total number of
5.2. EXPLORATORY STATISTICAL ANALYSIS

Table 5.4: Pearson correlation coefficients between ATL language metrics.

<table>
<thead>
<tr>
<th></th>
<th>nrl</th>
<th>nhp</th>
<th>nar</th>
<th>pelem</th>
<th>rbnd</th>
<th>col</th>
<th>cmp</th>
<th>lrfo</th>
<th>lrfi</th>
<th>hpfo</th>
</tr>
</thead>
<tbody>
<tr>
<td>nrl</td>
<td>0.92***</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nhp</td>
<td>0.91***</td>
<td>0.85***</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nar</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pelem</td>
<td>0.63***</td>
<td>0.47**</td>
<td>0.45**</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rbnd</td>
<td>0.69***</td>
<td>0.57***</td>
<td>0.48***</td>
<td>0.97***</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>col</td>
<td>0.96***</td>
<td>0.91***</td>
<td>0.84***</td>
<td>0.62***</td>
<td>0.71***</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmp</td>
<td>0.74***</td>
<td>0.69***</td>
<td>0.61***</td>
<td>0.60***</td>
<td>0.57***</td>
<td>0.68***</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lrfo</td>
<td>0.88***</td>
<td>0.86***</td>
<td>0.72***</td>
<td>0.68***</td>
<td>0.79***</td>
<td>0.92***</td>
<td>0.65***</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lrfi</td>
<td>0.53***</td>
<td>0.69***</td>
<td>0.43**</td>
<td>0.49***</td>
<td>0.60***</td>
<td>0.61***</td>
<td>0.45**</td>
<td>0.78***</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hpfo</td>
<td>0.37*</td>
<td>0.63***</td>
<td>0.20</td>
<td>0.18</td>
<td>0.31*</td>
<td>0.44**</td>
<td>0.39**</td>
<td>0.52***</td>
<td>0.74***</td>
<td></td>
</tr>
<tr>
<td>hpfi</td>
<td>0.71***</td>
<td>0.80***</td>
<td>0.54***</td>
<td>0.44**</td>
<td>0.56***</td>
<td>0.74***</td>
<td>0.59***</td>
<td>0.75***</td>
<td>0.71***</td>
<td>0.84***</td>
</tr>
</tbody>
</table>

Table 5.5: Pearson correlation coefficients between RSA language metrics.

<table>
<thead>
<tr>
<th></th>
<th>nmpd</th>
<th>mp</th>
<th>mpdfo</th>
<th>mpdfi</th>
<th>cmp</th>
</tr>
</thead>
<tbody>
<tr>
<td>nmpd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mp</td>
<td>0.87***</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mpdfo</td>
<td>0.90***</td>
<td>0.99***</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mpdfi</td>
<td>0.90***</td>
<td>0.99***</td>
<td>1.00***</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmp</td>
<td>0.60**</td>
<td>0.85***</td>
<td>0.83***</td>
<td>0.83***</td>
<td></td>
</tr>
<tr>
<td>col</td>
<td>0.75***</td>
<td>0.61**</td>
<td>0.66**</td>
<td>0.66**</td>
<td>0.57**</td>
</tr>
</tbody>
</table>

changes in the ATL dataset and the median in the RSA mapping dataset is 6 and 5, respectively. At $P < 0.01$, the test does not find a significant shift in the distribution of the number of ATL changes compared to the distribution of the number of RSA changes (Mann Whitney $U = 5423.5, n_{ATL} = 114, n_{RSA} = 80, P = 0.018$, two-tailed). Similarly, there is no significant difference for the total change size, calculated as total changes multiplied average change size (Mann Whitney $U = 3726, n_{ATL} = 114, n_{RSA} = 80, P = 0.029$, two-tailed). There is no evidence that the implementation choice of ATL or the RSA mapping framework affects the number of changes and overall size of changes per commit.
5.3 Exploratory Clustering Analysis

Unsupervised clustering techniques can be used to find structure and relationships in high-dimensional datasets. In this section, clustering analysis is employed to investigate whether there are structural differences in the datasets that might not be obvious via graphical inspection.

A hierarchical agglomerative (bottom up) clustering algorithm is selected, because the resulting output is a tree of clustering rules, called a dendrogram, that can be useful when interpreting cluster results. Additionally, sample size is small enough that hierarchical clustering’s relatively large time complexity cost of $O(n^2 \log n)$ is acceptable. Another advantage of the hierarchical clustering technique is that the number of clusters does not need to be specified to create the full dendrogram. After inspecting a generated dendrogram, it can be “cut” to a given tree height or to a given number of clusters.

The often used Euclidean distance metric is selected for determining the dissimilarity of individual observations, so all attributes are normalized by a z-transformation to address bias from different attribute scales. Also, starting from an initial assumption that the relative importance of attributes is equal, all attributes are assigned equal weighting. Feature reduction was considered, however the data has a low enough dimensionality that analysis complexity is not an issue. A future extension to this analysis may investigate the effect of feature selection on the quality of clusters. Finally the Complete Linkage Algorithm (CLA) is selected for the iterative merging of clusters to generate the dendrogram. Studies have shown that CLA works well for software systems [39].
5.3. EXPLORATORY CLUSTERING ANALYSIS

5.3.1 Language metric data vs. change metric data

The plots of dendrograms generated for ATL language metrics and ATL change metrics are presented in Figure 5.2. Working from the bottom of the tree upward, every leaf of the tree represents a cluster. A horizontal line segment above a group of clusters represents a merge of those clusters. Finally, the height of each horizontal line represents the dissimilarity of the clusters that it merges. Comparing the two dendrograms, we see that the change metrics produce more distinct clusters compared to the language metrics. The branches of the language metric dendrogram rapidly settle at a small level of dissimilarity compared to the branches of the change metrics. Further, the most dissimilar clusters are simply outliers (clusters that only contain single members), whereas membership in the change metric clusters is more evenly distributed.

When inspecting cluster membership, it is helpful to cut the dendrograms to restrict the number of clusters to a manageable level. The height of the cut depends on the number of clusters desired as well as the desired distribution of members in the clusters. In this case, the change metric dendrogram is cut at a height of 4.5, while the language metric dendrogram is cut at a height of 7. The height parameter is chosen based on a visual inspection of the dendrogram.

Table 5.6 presents the cluster membership along with some simple observations about the cluster members. Though there are some singleton clusters, most of the observations fall into two major clusters. Change steps in cluster 2 tend to have smaller size changes and are at a lower depth compared to change steps in cluster 1. Interestingly, clusters 3 and 4 seem to have a large proportion of refactoring and corrective changes.
5.3. EXPLORATORY CLUSTERING ANALYSIS

Figure 5.2: Dendrograms for (a) ATL change metrics and (b) ATL language metrics
5.3. EXPLORATORY CLUSTERING ANALYSIS

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Members</th>
<th>Observations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.1, 1.2, 1.2, 1.2C1, 1R.1, 2.4, 2.1, 2.2, 2.3, 2.4C2, 3.2, 4.3, 3.5, 5.6, 5.7+5.8, 5.11, 2.3C3, 5.9C1</td>
<td>Main cluster</td>
</tr>
<tr>
<td>2</td>
<td>1.3, 1.3, 4.3+4.4, 4.2, 4.3C1, 5.1, 5.2, 5.3, 5.4, 5.10</td>
<td>Similar composition of change types as main cluster, smaller size changes, deeper in tree</td>
</tr>
<tr>
<td>3</td>
<td>1.3C1, 2.4C1, 2R.1, 3.1+3.4, 3.6, 4.4C1</td>
<td>Small refactoring/corrective changes, smaller than cluster 4 changes</td>
</tr>
<tr>
<td>4</td>
<td>2.3C1, 4.3C2, 2.3C2</td>
<td>Similar depth, same composition of change operations, all corrective changes</td>
</tr>
<tr>
<td>5</td>
<td>3.3, 4.1, 4R.2</td>
<td>Many changes, wide range of depth</td>
</tr>
<tr>
<td>6</td>
<td>4R.1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5.5</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>5.9</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>5.11C1</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.6: Cluster membership for ATL change metrics dendrogram (h=4.5)

Table 5.7 shows the cluster membership for change steps, using ATL language metrics data. The change steps tend to be grouped simply in terms of the size of the transformation artifact. Additionally, the change steps are grouped by their corresponding milestones. For example cluster 2 contains mostly milestone 1 changes, whereas cluster 4 contains mostly milestone 2 changes.

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Members</th>
<th>Median rule count (MAD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0 (0)</td>
</tr>
<tr>
<td>2</td>
<td>1.1, 1.2, 1.3, 1.2, 1.3, 1.2C1</td>
<td>2.5 (0.74)</td>
</tr>
<tr>
<td>3</td>
<td>1R.1</td>
<td>3 (0)</td>
</tr>
<tr>
<td>4</td>
<td>2.4, 2.1, 2.2, 2.3, 1.3C1, 2.4C1, 2.4C2, 2.3C1, 2R.1, 3.1+3.4, 3.2</td>
<td>4 (0)</td>
</tr>
<tr>
<td>5</td>
<td>3.3, 4.1, 4.3, 4.3+4.4, 4.2, 4.3C1, 4.3C2, 2.3C2, 3.6, 3.5, 4.4C1, 4R.1, 4R.2, 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.7+5.8</td>
<td>12 (0)</td>
</tr>
<tr>
<td>6</td>
<td>5.9, 5.10, 5.11, 2.3C3, 5.11C1, 5.9C1</td>
<td>14 (0)</td>
</tr>
</tbody>
</table>

Table 5.7: Cluster membership for ATL language metrics dendrogram (h=7), including corresponding rule count median and median absolute deviation.
5.3. EXPLORATORY CLUSTERING ANALYSIS

5.3.2 RSA mapping tool data sets

A set of clusters is also generated for the RSA mapping tool change metrics data and language metrics data. The resulting dendrograms are shown in Figure 5.3. The RSA change metric dendrogram is cut at a height of 6.1, while the RSA language metric dendrogram is cut at a height of 3. Clusters generated using RSA language metrics are similar to those generated with ATL language metrics; they simply group the change steps by the size of transformation artifacts and the corresponding milestones.

Observations for the RSA change metric clustering results are listed in Table 5.8, while RSA language metric clustering membership is listed in Table 5.9.

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Members</th>
<th>Observations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.1+1.3, 3.1, 3.2, 3.6, 4.4</td>
<td>Dominated by create operations</td>
</tr>
<tr>
<td>2</td>
<td>1.2+1.3, 2.4C2, 4.1</td>
<td>Large number of separate changes, split between create and update operations</td>
</tr>
<tr>
<td>3</td>
<td>2.4, 3.5+3.3, 3.3C1, 4.2</td>
<td>Few very large sized changes (process definitions, condition and listen branches)</td>
</tr>
<tr>
<td>4</td>
<td>2.3, 1.3C1</td>
<td>Update heavy, deep in tree, lots of diversity (expressions)</td>
</tr>
<tr>
<td>5</td>
<td>2.4C1, 2.1, 2.2, 3.4, 4.3</td>
<td>Small, contained changes, low height</td>
</tr>
<tr>
<td>6</td>
<td>3.2C1</td>
<td>Only member that contains move operations</td>
</tr>
</tbody>
</table>

Table 5.8: Cluster membership for RSA change metrics dendrogram (h=6.1)

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Members</th>
<th>Median rule count (MAD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0 (0)</td>
</tr>
<tr>
<td>2</td>
<td>1.1+1.3, 2.4</td>
<td>5 (4.5)</td>
</tr>
<tr>
<td>3</td>
<td>1.2+1.3</td>
<td>6 (0)</td>
</tr>
<tr>
<td>4</td>
<td>2.3, 2.4C1, 2.1, 2.2, 3.1, 3.2, 3.4, 3.5+3.3, 3.6, 3.3C1</td>
<td>11 (3.0)</td>
</tr>
<tr>
<td>5</td>
<td>2.4C2, 4.1, 4.2, 4.3, 4.4, 1.3C1, 3.2C1</td>
<td>20 (0)</td>
</tr>
</tbody>
</table>

Table 5.9: Cluster membership for RSA language metrics dendrogram (h=3.0), including corresponding rule count median and median absolute deviation.
5.3. EXPLORATORY CLUSTERING ANALYSIS

Figure 5.3: Dendrograms for (a) RSA change metrics and (b) RSA language metrics
5.4 Classification Analysis

Supervised learning techniques are often used to build predictive models, so that a target attribute’s value can be predicted based on a given set of input attributes. This section shows how the different metrics can be used to build a simple classifier. The classification model is built from data at the change step granularity.

Each observed change in the dataset is assigned a class. Change classes are based on the original categories of software maintenance proposed by Lientz, Swanson, and Tompkins [37]:

1. **adaptive**: representing changes made to support new or modified requirements,

2. **corrective**: representing changes made to correct defects found through testing or manual inspection, and

3. **perfective**: representing changes made to refactor the elements.

A decision tree classification algorithm called rpart, available in R via the rpart package, is selected to build the classifier models. The implementation of rpart is based on the Classification and Regression Tree (CART) algorithm [36]. Decision tree classifiers tend to be more easily interpretable compared with other types of classifier models because the decision splits are formed directly from the original values of the attributes. Decision trees based on the CART algorithm are also non-parametric, and therefore can be more robust for non-normal data. It should be reiterated that the purpose of this analysis is not to build a robust and well optimized final model, but to compare different types of metrics on a classification problem.

The leave-one-out cross validation method is used to perform validation and the cross-validation error rates are reported. In this case, leave-one-out cross validation
5.4. CLASSIFICATION ANALYSIS

is feasible due to a small sample size.

5.4.1 ATL Change Metrics Classifier Results

Figure 5.4 shows the results of the training and cross-validation for ATL change metrics data. Note that while developing the model, a better selection of attributes was found by only considering the descriptive change attributes. The optimal number of splits in the tree occurs at 3, after which the simulated test error increases. Average depth and average size are the attributes that are selected once the tree is pruned.

To navigate the tree, start from the root node. If the attribute of the observation satisfies the boolean expression, then navigate down the left branch. Otherwise
navigate down the right branch. Continue until a leaf node is reached. The estimated misclassification error rate for this model is 27.2%. Further evaluation of the classifier models would involve gathering predictions from a completely separate test data set. The results could then be compared using, for example, Receiver Operating Characteristic curves.

5.5 Discussion of Results

From the initially defined set of language metrics, it was found that only a subset of language metrics were actually relevant to the specific transformation task. The generic change metrics are calculated at the abstract syntax level and are based on the Ecore representation of the model artifacts, therefore they are not affected as much by the usage of specific language features. This suggests that the generic metrics based on the Ecore representation can be used as a starting point for language-specific metric selection, if experience is lacking for the measurement of the transformation task or the transformation language.

The cluster membership of changesteps using ATL language metrics tend to be grouped simply in terms of the size of the transformation artifact and as well the corresponding milestone of the changesteps. Clusters generated using RSA language metrics show a similar result. This is perhaps not very surprising since most of the language metrics are related in some way to artifact size, which increases over time due to the constructive nature of the transformation task.

In contrast, cluster membership for the change metrics data appears to reflect the types and intent of individual changes more clearly than for the language metrics data. For these particular implementations of the transformation task, the change
5.6. THREATS TO VALIDITY

metrics can identify patterns and structure in historical data that are not necessarily detected using language metrics.

One notable difference between RSA and ATL is that the depth metric varies more in the RSA change metrics clusters. Change depth appears to play a larger role in determining cluster membership for RSA. This could be a result of less integration between the expression language (Java) and the declarative mapping. In the ATL language artifacts, OCL expressions are used at every level so the average depth of changes may not be as distinct between expressions and mappings.

5.6 Threats to validity

Conclusions from this specific demonstration are limited, due to the small amount of data available for analysis. Ideally, an empirical study would involve several transformation projects over long periods of time, implemented by many different developers. There is a strong possibility of learning bias because the artifacts from both languages are developed by the same developer. A further contribution to bias is that, although the path of development of the transformation is “recorded” in the mapping specification, the actual transformation is implemented with prior knowledge of the change analysis. Also specifically for this experiment, the method to commit the data over several implementations of the transformation is somewhat artificial, due to the nature of the fine-grained manual commits. This may have caused an additional bias in the way changes were made to the transformation artifacts.

Another potential threat lies in the creation of difference models. Model representations of the transformation language artifacts do not necessarily include unique identifiers for model elements. ATL employs a parser to build an abstract syntax
tree and the parser cannot maintain unique identifiers across these separate versions. The RSA mapping tool does not maintain identifiers even though its artifacts are based on an XML-based format. In the absence of model element identifiers, EMF Compare must heuristically match elements between versions. The difference model is then based on this match model. If the match phase does not provide a correct input for the difference phase, then anomalies may appear in the difference model.

Also, metrics calculated from different abstract representations may not be directly comparable. Reasoning and statistical normalization techniques must then be applied to determine if the metrics can be interpreted to support comparison. This is only a concern for the subset of analysis that involves language comparison.

Finally the choice of metrics can always be argued. In this case the analysis may have benefited from a larger choice of language metrics. However lack of time nearing the end of the research prevented further iterations of the data mining process.
Chapter 6

Summary and Conclusions

6.1 Summary

A preliminary set of metrics is proposed to measure the amount of change activity in a modelling language artifact using a graph-based difference model of the artifact. The work presented here also introduces a process as a foundation for applying MSR-based exploratory analysis to iterative model transformation implementations. The process is demonstrated on an incrementally defined transformation task, considering two different model transformation languages.

This process can be applied to any language (graphical/textual) as long as its abstract syntax can be modelled and its artifacts can be represented in a model form. However some issues were also encountered in the data gathering phase because of the graph-based nature of the difference model. As well, the transformation languages chosen for the demonstration are both declarative mapping languages so the general validity of applying this process for different types of languages is an open question.

Language expertise may be difficult to find when considering new modelling languages and smaller domain specific languages. Exploratory analysis with change
metrics can inform users and designers of the language about change patterns that may be worth investigating further. General change-related metrics have value in this scenario because they reduce the need to consider the features and the semantics of the language during data gathering.

6.2 Future Directions

In the future it would be interesting to explore more transformation tasks and, given enough useful data, perform proper empirical experiments. This would support a better understanding of the incremental development and evolution of artifacts in Model-Based Engineering projects using historical data. An open repository of data, including commit history and even issue tracking data, would be of immense benefit. Repositories and various transformation “zoos” are a step in this direction, but they do not currently provide change history.

Extensions of the change model used in the extraction process are another interesting direction. The consideration of data repositories containing information such as issue tracking reports and mailing list emails could provide more opportunities for analysis, just as it has in the MSR field. Additionally, this information may be useful for increasing automation when detecting change step boundaries.

Another direction to consider is the relationship of change metrics with quality attributes like reliability and understandability. The inclusion of other types of repositories might help to increase correlation between certain metrics and these additional quality attributes. For example metrics for issue tracking repositories may correlate well with reliability. This type of analysis may compliment existing work to measure static model transformations using metrics in [1], providing additional validation of
the correlation between static metrics and quality properties.

Finally, a more comprehensive model of change integrated with existing modelling tools would assist the analysis of data. This change model could incorporate elements of the mining process, metrics, quality properties, and sequential change data. If, for example, such a change model is realized in EMF, then existing Eclipse modelling tools can be leveraged in data preparation and even some analysis.

These future directions for research represent steps toward the original motivating long-term goal of using historical software development data and transformation artifact change models to aid in the model transformation development process.
Bibliography


