An Extensible Platform for Specification of
Integrated Languages for Model Management

Dimitrios Kolovos
Submitted for the degree of Doctor of Philosophy

The University of York
Department of Computer Science
June 2008
Abstract

This thesis addresses the problem of integrated and uniform programmatic model management in Model Driven Engineering (MDE). An MDE process typically involves a number of models that are expressed using different modelling languages and technologies. In the context of an MDE process, many different operations such as validation, comparison, model-to-model transformation, model-to-text transformation, refactoring and merging need to be performed on such models in an automated manner. The hypothesis of this thesis is that these tasks can be supported in an integrated and uniform way by a platform that provides reusable core facilities and an extensible architecture that enables the construction of interoperable and consistent task-specific languages that share abstract and concrete syntax, operational semantics and implementation. To evaluate the validity of the hypothesis, such a platform has been designed and a reference implementation has been provided. Results obtained from relevant case studies in the ModelWare and ModelPlex EU projects as well as in the context of internal and external collaborations validate the hypothesis as they demonstrate the significant reuse, interoperability and uniformity benefits of such an approach.
# Contents

1 Introduction .......................... 16
   1.1 Motivation .......................... 17
      1.1.1 Models in Model Driven Engineering .......................... 17
      1.1.2 Model Management .......................... 18
      1.1.3 Coordination and Orchestration .......................... 19
      1.1.4 Summary .......................... 20
   1.2 Hypothesis and Objectives .......................... 21
   1.3 Research Methodology .......................... 22
      1.3.1 Analysis .......................... 22
      1.3.2 Iterative Design and Implementation .......................... 23
      1.3.3 Iterative Testing .......................... 24
   1.4 Research Results .......................... 24
   1.5 Thesis Structure .......................... 25

2 Literature Review .......................... 29
   2.1 Model Driven Engineering .......................... 29
   2.2 Software Models .......................... 30
   2.3 Metamodelling Architectures .......................... 31
      2.3.1 Meta-Object Facility (MOF) .......................... 32
      2.3.2 Eclipse Modeling Framework (EMF) .......................... 33
      2.3.3 Microsoft DSL Tools .......................... 34
   2.4 Model Management Tasks and Tools .......................... 34
      2.4.1 Model Validation .......................... 34
      2.4.2 Model Transformation .......................... 43
2.4.3 Model Comparison ........................................ 48
2.4.4 Model Merging ........................................... 51
2.4.5 Model-to-Text Transformation ..................... 54

2.5 Interoperability, Reuse and Consistency of Model Management Languages .................. 54
2.5.1 Integration and Interoperability .................... 55
2.5.2 Code Consistency and Reuse ....................... 58
2.5.3 Interoperability, Reuse and Consistency in Contemporary Model Management Languages ....... 59
2.5.4 Platforms of Model Management Languages .... 59

2.6 Chapter Summary ........................................ 60

3 Analysis and Hypothesis ................................. 62
3.1 Research Challenges ..................................... 62
3.2 Research Hypothesis ..................................... 65
3.3 Research Scope .......................................... 67
3.4 Research Methodology ................................. 67
3.4.1 Analysis .............................................. 67
3.4.2 Iterative Design and Implementation ............. 68
3.4.3 Iterative Testing ..................................... 69
3.5 Chapter Summary ........................................ 69

4 Architecture .................................................. 70
4.1 Quality Attributes ....................................... 71
4.1.1 System quality attributes ........................... 71
4.1.2 Business quality attributes ....................... 72
4.2 Architectural Tactics ................................... 72
4.2.1 Maintenance of semantic coherence .............. 72
4.2.2 Separate interface from implementation ......... 73
4.2.3 Information hiding .................................. 73
4.2.4 Deferring binding time ............................. 73
4.3 Architectural Overview ................................. 73
4.4 The Model Connectivity Layer ....................... 74
6.1.2 Concrete Syntax .......................... 130
6.1.3 Execution Semantics ...................... 131
6.1.4 Fuzzy and Dictionary-based String Matching 135
6.1.5 Interactive Matching ...................... 136
6.1.6 Exploiting the Comparison Outcome ........ 137

6.2 The Epsilon Transformation Language (ETL) .......... 138
6.2.1 Style .................................. 138
6.2.2 Source and Target Models ................. 139
6.2.3 Abstract Syntax ......................... 139
6.2.4 Concrete Syntax .......................... 141
6.2.5 Execution Semantics ...................... 142
6.2.6 Interactive Transformations ................ 146
6.2.7 Summary ................................ 147

6.3 The Epsilon Validation Language (EVL) .......... 148
6.3.1 Motivation ................................ 148
6.3.2 Abstract Syntax ......................... 152
6.3.3 Concrete Syntax .......................... 155
6.3.4 Execution Semantics ...................... 156
6.3.5 Intra-Model Consistency Checking Example .... 158
6.3.6 Inter-Model Consistency Checking Example .... 164
6.3.7 Summary ................................ 169

6.4 The Epsilon Wizard Language (EWL) .......... 171
6.4.1 Motivation ................................ 171
6.4.2 Update Transformations in the Small ........... 173
6.4.3 Abstract Syntax ......................... 175
6.4.4 Concrete Syntax .......................... 177
6.4.5 Execution Semantics ...................... 177
6.4.6 Examples ................................ 177
6.4.7 Summary ................................ 184

6.5 The Epsilon Merging Language (EML) .......... 184
6.5.1 Motivation ................................ 185
6.5.2 Realizing a Model Merging Process with Epsilon 186
6.5.3 Abstract Syntax .......................... 187
List of Figures

1.1 Overview of the research methodology .......................... 23
2.1 A Typical Metamodeling Architecture ......................... 32
2.2 Task Dependencies ............................................. 56
3.1 Overview of the research methodology ......................... 68
4.1 Overview of the architecture of Epsilon ....................... 75
5.1 Overview of the Epsilon Model Connectivity layer ........... 85
5.2 EOL Module Structure ......................................... 92
5.3 Overview of the type system of EOL .......................... 100
5.4 Overview of the feature navigation EOL expressions ....... 109
5.5 The Tree Metamodel ............................................ 121
5.6 Example of an Eclipse-based IUserInput implementation ... 123
5.7 Example of a command-line-based IUserInput implementation 124
6.1 ECL Abstract Syntax ............................................ 129
6.2 ECL Match Trace ................................................ 132
6.3 The Tree Metamodel ............................................. 134
6.4 ETL Abstract Syntax ............................................ 140
6.5 A Simple Graph Metamodel .................................... 144
6.6 ETL Runtime .................................................... 145
6.7 Abstract Syntax of EVL ........................................ 153
6.8 The ProcessLang Metamodel ................................... 165
6.9 The ProcessPerformanceLang Metamodel ...................... 165
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Consistency Classification</td>
<td>38</td>
</tr>
<tr>
<td>5.1</td>
<td>Operations of type Any</td>
<td>98</td>
</tr>
<tr>
<td>5.2</td>
<td>Operations of type String</td>
<td>100</td>
</tr>
<tr>
<td>5.3</td>
<td>Operations of type Real</td>
<td>102</td>
</tr>
<tr>
<td>5.4</td>
<td>Operations of type Collection</td>
<td>102</td>
</tr>
<tr>
<td>5.5</td>
<td>Operations of types Sequence and OrderedSet</td>
<td>104</td>
</tr>
<tr>
<td>5.6</td>
<td>First-order logic operations on Collections</td>
<td>105</td>
</tr>
<tr>
<td>5.7</td>
<td>Operations of type Map</td>
<td>105</td>
</tr>
<tr>
<td>5.8</td>
<td>Operations of Model Element Types</td>
<td>107</td>
</tr>
<tr>
<td>5.9</td>
<td>Arithmetical operators</td>
<td>110</td>
</tr>
<tr>
<td>5.10</td>
<td>Comparison operators</td>
<td>110</td>
</tr>
<tr>
<td>5.11</td>
<td>Logical Operators</td>
<td>111</td>
</tr>
<tr>
<td>5.12</td>
<td>Implies Truth Table</td>
<td>111</td>
</tr>
<tr>
<td>5.13</td>
<td>Default values of primitive types</td>
<td>112</td>
</tr>
<tr>
<td>5.14</td>
<td>Operations of IUserInput</td>
<td>122</td>
</tr>
<tr>
<td>9.1</td>
<td>Matrix of Tasks and applied Languages</td>
<td>257</td>
</tr>
<tr>
<td>9.2</td>
<td>Lines of Java Code for each Language Runtime</td>
<td>272</td>
</tr>
<tr>
<td>9.3</td>
<td>Lines of Grammar and Generated Code for each Language Runtime</td>
<td>272</td>
</tr>
<tr>
<td>9.4</td>
<td>Lines of Hand-Crafted Grammar and Code for each Language Runtime</td>
<td>273</td>
</tr>
<tr>
<td>9.5</td>
<td>Lines of Code Required for the Development Tools</td>
<td>273</td>
</tr>
<tr>
<td>9.6</td>
<td>Total Lines of Code Required for Supporting each Language</td>
<td>274</td>
</tr>
</tbody>
</table>
9.7 Supported Tasks Matrix
Acknowledgements

I am most grateful to my supervisors, Prof. Richard Paige and Dr. Fiona Polack for their invaluable support and feedback throughout the duration of this degree, and all my co-authors and colleagues for the fruitful discussions that have contributed to shaping the views reflected in this thesis.

I would like to express my warmest gratitude to my parents Stavros and Stamatia Kolovos, my partner Jenny Orfanou, and all my friends for the support they have offered me throughout those years.

I would like to thank my external examiners, Prof. Jean Bezivin and Prof. Jon Whittle, and my internal examiner, Dr. Neil Audsley, for their insightful comments and feedback which have greatly improved this report.

I am indebted to Prof. Theodore Kalamboukis for the guidance and opportunities he has offered me, Marios Sintichakis for being a true inspiration for me becoming a software engineer, and Dr. Yannis Zorgios for providing me with opportunities, guidance and feedback without which I would have most probably never got into Model Driven Engineering in the first place.

Finally, this work would not have been possible without the financial support of the European Commission via the ModelWare and ModelPlex EU projects, co-funded under the “Information Society Technologies” Sixth Framework Programme (2002-2009).
Author Declaration

Parts of this work have been previously published by the author in a number of journals, international conferences and workshops. Publications [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19] represent original work by the author. The author has also contributed to [20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31]. Additional details about the nature of the contributions of the author to the latter are provided in Section 9.2.2 of this thesis.
Chapter 1

Introduction

In the continuous process of improving the quality of software and the productivity of those who develop and maintain it, various methodologies and techniques have been proposed and adopted over the years. While approaches vary significantly, the vast majority share a common vision: that of raising the level of abstraction at which software is designed and implemented.

Therefore, from primitive assembly languages, engineers have moved on to structured (e.g. Fortran, C), object \[32\] (e.g. C++, Java, .NET) and aspect \[33\] (e.g. AspectJ) oriented programming. Other examples of this trend include evolution from machine-specific code to machine-independent code and even to operating-system independent (e.g. Java, .NET) code. As well, there has been the replacement of flat data-storage files with sophisticated Database Management Systems that provide an increased level of abstraction for interacting with stored data. Designing and implementing software at a high level of abstraction allows engineers to suppress inessential details and focus on more important aspects of the system thus enhancing both their productivity and the quality of the product.

Model Driven Engineering (MDE) is a state of the art approach to software development that attempts to raise the level of abstraction at which software and system engineers carry out their tasks \[34\] beyond the application of programming languages and the use of components. The proposed
way to achieve this is through promoting models into first-class artefacts of the development process. The basic principle of MDE is that models should be precisely specified and successively refined until they reach a level of maturity and completeness where they can be automatically transformed into the actual system.

This chapter presents a brief overview of the current state of practice in Model Driven Engineering and highlights the problems that motivated the work in this thesis. Following that it outlines the research hypothesis and methodology and provides a summary of the results and the main contributions of the thesis. Finally, it provides an overview of the organization of the thesis and a summary of the remaining chapters.

1.1 Motivation

This section presents a summary of the current state of practice in Model Driven Engineering and highlights the problems that motivated the work in this thesis.

1.1.1 Models in Model Driven Engineering

In its most abstract definition, a model is a description of a phenomenon of interest [35]. In general, models can be characterized as structured or unstructured. Unstructured models are descriptions that do not adhere to well-defined syntactic or semantic rules. Examples of this type of models are board drawings, informal sketches, natural language specifications etc. Due to their lack of conformance to well-defined rules, unstructured models are only useful for communication between humans. By contrast, a structured model is an artefact that conforms to a set of syntactic and semantic well-formedness constraints which is called its modelling language or metamodel. Due to their rigorously defined nature, except for human communication purposes, structured models are also useful for mechanised processing.

Two types of modelling languages are generally recognized: general-purpose and domain-specific [36]. General purpose modelling languages
(such as the Unified Modeling Language [37] and the Z formal specification language [38]) target a wide spectrum of problems and thus provide generic constructs that can be used to model systems across diverse domains. Domain specific languages (DSLs) generally target a narrow range of problems for which they provide tailored constructs. In a broader context, other types of languages such as programming languages (e.g. Java, C#) or meta-data languages (e.g. XML) can also be viewed as modelling languages in the sense that they provide a well-defined syntax and semantics that abstracts away from lower level constructs (e.g. JVM instructions for Java, ASCII for XML).

In the context of an MDE process more than one general purpose and/or domain specific modelling languages, styles or paradigms may be used to capture different aspects of the designed system at different levels of abstraction. As a result, an MDE workspace typically involves a number of inter-related, and potentially overlapping models captured using different modelling languages and/or diverse modelling technologies.

1.1.2 Model Management

To deliver its proposed benefits, MDE inherently relies on the existence of mechanisms that can automate a range of model management tasks such as model validation, model-to-model transformation and model-to-text transformation. For instance, to realize the benefits of increased productivity in software construction, mechanisms that can automatically generate programming-language source code from more abstract models should be available. As another example, to interchange models between different modelling languages/technologies without much effort, it is important that automated model transformation mechanisms with such capabilities are provided. Also, to maintain consistency between several loosely coupled models, automated inconsistency detection mechanisms are necessary.

The complexity of model management tasks is typically such that complex mechanisms are required to automate them. The current widely accepted view is that each model management task is better supported by a
corresponding task-specific programming language that embraces its individual requirements and characteristics. As a consequence, a number of task-specific languages for tasks such as model-to-model transformation, model validation and model-to-text transformation have been proposed. By contrast, as identified through the review performed in Chapter 2, no suitable task-specific languages exist for tasks such as model comparison and merging, inter-model validation and in-place model transformation. Thus, in addition to the number of heterogeneous models discussed in Section 1.1.1, a typical MDE workspace also contains a number of model management programs implemented using different task-specific languages.

The majority of contemporary model management languages have been developed independently of each other. As a consequence, as discussed in detail in Section 2.5, they demonstrate significant duplication as similar features have been designed and implemented many times by different development teams. More significant than duplication though is the problem of inconsistency: different language developers have designed and implemented similar features in similar but not consistent ways, thus rendering inter-language reuse and interoperability particularly challenging.

1.1.3 Coordination and Orchestration

An MDE process typically consists of a number of model management (e.g. model to model transformation, model validation) and classical software development (e.g. code compilation, file system management) tasks. For example, a simplified MDE process for constructing a data-intensive object-oriented application starting from a UML model typically involves the following steps:

1. Validate the UML model
2. Generate Java code from the UML model
3. Transform the UML model into a Database model
4. Generate SQL code from the Database model
5. Compile the generated Java code

6. Deploy the SQL code into a DBMS

The first step of this scenario involves validating the source UML model in order to identify inconsistencies that will impact the remaining steps of the process. If the UML model is found to be consistent, the next step is to generate Java code from it using a model-to-text transformation. Following that, SQL code for persisting/retrieving the data managed by the application needs to be generated from the UML model. This is achieved in two steps: in step 3, the UML model is transformed into a model that conforms to a domain specific language tailored to describing Database structures and in step 4 this intermediate Database model is transformed into SQL code using a model-to-text transformation. Finally, in step 5 the generated Java code needs to be compiled and the SQL code needs to be deployed in the underlying Database Management System (DBMS).

Summarizing the above, this simple scenario involves two different modelling languages (UML, Database DSL) - one of which is domain specific - three model management tasks (validation, model-to-model transformation and model-to-text transformation) and two non-MDE tasks (code compilation and deployment).

Given the number of modelling languages and technologies, the number of different task-specific languages used to implement individual model management tasks, as well as the need to coordinate MDE with non-MDE tasks and the potentially complex dependencies between different tasks, an automated process coordination and orchestration mechanism is of paramount importance.

1.1.4 Summary

An MDE process involves a number of models expressed using different modelling languages and technologies as well as a number of model management programs implemented using task-specific languages which are predominantly inconsistent with each other and as such they provide limited
opportunities for reuse and interoperability. Moreover, a number of tasks such as model comparison, merging and user-driven in-place model transformation are not currently supported by task-specific languages.

1.2 Hypothesis and Objectives

With respect to the current situation discussed above, the context of the research hypothesis is as follows:

A Model Driven Engineering process can involve many different model management tasks such as validation, transformation (including model-to-model and model-to-text), in-place model transformation, comparison and merging. Currently, there are a number of independently developed task-specific languages that support some of these tasks, particularly model validation, model-to-model and model-to-text transformation. By contrast, no task-specific languages have been proposed for tasks such as model comparison, merging, inter-model validation and in-place transformation. Also, the fact that existing task-specific languages have been predominantly developed independently of each other leads to consistency, reuse and interoperability problems.

From the language engineering point of view, when constructing a new language in the absence of an underlying framework that provides a set of reusable common facilities, engineers need to reimplement a significant amount of functionality, and experience has demonstrated that this is typically done in an inconsistent manner that results in uniformity and interoperability problems between the new language and existing languages for other model management tasks. From the user’s point of view, the user needs to learn and work concurrently with a number of inconsistent and isolated languages that do not interoperate with each other, and across which code cannot be reused and therefore needs to be duplicated and maintained separately.

In this context, the hypothesis of this thesis is stated as follows:

*Despite their individual requirements and characteristics, a wide range
of task-specific modelling languages share a significant number of common features, and therefore, instead of developing each language separately, it is beneficial in terms of reuse, uniformity and interoperability, both from a language engineering and a user perspective, to develop them atop a platform that provides a reusable set of commonly required features.

The objectives of the thesis are:

1. To develop a platform atop which uniform, interoperable and reusable languages can be developed.

2. To use the platform to develop task-specific languages that address the, now largely unsupported, tasks of inter-model consistency checking, model comparison, model merging and in-place model transformation.

3. To use the platform to develop uniform task-specific languages for tasks that are already supported by existing languages (e.g. model-to-model transformation, model validation).

4. To develop an orchestration and coordination framework that enables composition of individual model management tasks implemented using languages of the platform into coherent workflows.

1.3 Research Methodology

To evaluate the validity of the hypothesis, a typical software engineering process involving an initial analysis phase, followed by several design, implementation and testing iterations has been followed. A graphical overview of the process appears in Figure 1.1. This section provides an overview of the main phases of the process.

1.3.1 Analysis

In the analysis phase, the most widely used metamodelling architectures that enable engineers to define modelling languages and models, and the
the most common model management tasks were identified. For each model management task, the state-of-the-art approaches in terms of supporting languages and tools were reviewed in order to identify their advantages and shortcomings. Through the analysis, a number of research challenges have been identified that have motivated the hypothesis and objectives of the thesis.

1.3.2 Iterative Design and Implementation

Following the analysis phase, a conceptual architecture has been conceived to investigate the hypothesis. This conceptual architecture has been refined into a technical design which in turn has guided the construction of the reference implementation.

In the first phase of design, infrastructure was designed based on the findings of the analysis. Following this, languages for previously largely unsupported tasks (model comparison and merging) were built atop the infrastructure. Through the process of building these first task-specific languages, the design and implementation of the infrastructure have been progressively improved and become more flexible in order to accommodate the needs of
the different tasks. Latterly, languages for model-to-model transformation, model validation and in-place model transformation were designed and implemented.

Finally, a workflow mechanism was designed and a prototype was implemented to enable orchestration and coordination of tasks implemented using different task-specific language of the platform.

1.3.3 Iterative Testing

Throughout the design and implementation iterations, several case studies have been used to assess the quality and usefulness of the proposed approach and the correctness of the reference implementation. Also significant feedback has been provided by academic peers who have reviewed a number of publications on several core aspects of the design of the platform and the languages that have been built atop it, and by external users who have been using the reference implementation. Errors and omissions identified during testing iterations were provided as input to further design and implementation cycles.

1.4 Research Results

As a result of this thesis, a layered architecture for specifying consistent, interoperable and reusable task-specific model management languages that share abstract and concrete syntax as well as operational semantics and implementation has been proposed, and a reference implementation, called Epsilon, has been constructed. The proposed architecture provides an abstraction layer for managing models of diverse modelling technologies, a feature-rich model navigation and modification language that can be reused by task-specific languages and an orchestration mechanism for seamlessly integrating individual model management tasks into coherent workflows.

The thesis has also contributed four novel languages that address previously largely unsupported tasks; namely model comparison, model merging, inter-model validation and user-driven in-place model transformation. It
has also proposed novel extensions in the field of model validation, such as constraint dependency management and inconsistency repairing facilities, and in the field of model-to-model transformation, such as interactive model transformations.

The hypothesis has been validated by demonstrating that a wide range of interoperable and consistent task-specific languages can be constructed atop the proposed architecture with minimal duplication and enhanced reuse. Also, a number of case studies performed in the context of the ModelPlex [39] and ModelWare [40] EU projects and other collaborations has demonstrated the practicality and usefulness of the architecture and the individual languages proposed in this thesis.

1.5 Thesis Structure

In Chapter 2 a detailed review of the related work is performed. The review is further structured into four sections. Section 2.1 discusses the rationale and benefits of Model Driven Engineering. Section 2.2 presents the different types of models involved in a software process and identifies the characteristics that models should possess in order to be suitable for use in a Model Driven Engineering process. Section 2.3 presents an overview of the most widely-used technologies and architectures that enable developers to define modelling languages and models suitable for MDE. Section 2.4 identifies the main model management tasks involved in MDE processes and presents the state-of-the-art approaches in terms of supporting languages and tools for each one of them. Finally, Section 2.5 presents a discussion on the aspects of integration, reuse and interoperability between languages that target different model management tasks.

Chapter 3 summarizes the findings of the review performed in Chapter 2 and identifies the shortcomings of contemporary approaches to model management. More specifically, Section 3.1 identifies shortcomings in terms of consistency, interoperability and reuse between model management languages that target different model management tasks. Following that, in Section 3.2 the research hypothesis is stated and a set of objectives for val-
idating the proposed hypothesis is outlined. In Section 3.4, the research methodology that is followed in order to evaluate the validity of the hypothesis is presented.

Chapter 4 outlines the architecture of an integrated family of model management languages named the Extensible Platform for Specification of Integrated Languages for mOdell maNagement (Epsilon) that is used as a means of exploring the research hypothesis. Sections 4.1 and 4.2 present the quality attributes that the architecture is designed against and the tactics that are employed to achieve them. Section 4.3 provides an overview of the organization of the architecture into layers. Section 4.4 introduces the Epsilon Model Connectivity (EMC) layer, a layer that aims at abstracting away from implementation diversities among different model management technologies and at providing a uniform interface for accessing models to the layers that build atop it. Section 4.5 provides an overview of the Epsilon Object Language (EOL) layer. EOL is an imperative OCL-based language that aims at providing a reusable set of mechanisms for navigating, querying, modifying and otherwise programmatically interacting with models. Section 4.6 outlines a set of task-specific languages that need to be built atop EMC and EOL to address the most recurring model management tasks. Section 4.7 outlines the workflow layer that is responsible for coordinating model management tasks implemented with different languages of the Epsilon platform, as well as with non-MDE tasks, to form complex workflows. Finally, Section 4.8 outlines the architecture of the end-user development tools that enable users to compose and execute model management programs.

Chapters 5, 6 and 7 refine the architectural overview outlined in Chapter 4 into a detailed design. More specifically, Chapter 5 presents the design details for the model connectivity layer and elaborates the abstract and concrete syntax as well as the execution semantics and novel features of the core model navigation and modification language of the platform.

Following that, Chapter 6 presents a set of task-specific model management languages that build on the provided infrastructure. Section 6.1 presents the Epsilon Comparison Language (ECL), a language for comparing and establishing correspondences between elements of different models.
Section 6.2 presents the Epsilon Transformation Language (ETL), a hybrid model-to-model transformation language. Section 6.3 presents the Epsilon Validation Language (EVL) a hybrid language for detecting and repairing both intra and inter-model inconsistencies. Section 6.4 presents the Epsilon Wizard Language (EWL), a language for interactive in-place model transformation and Section 6.5 presents the Epsilon Merging Language (EML), a language for merging models based on given correspondences (established using ECL or otherwise). In each section, the abstract and concrete syntax and the execution semantics of each language are presented with a focus on the novel features provided by the language compared to existing languages (where applicable) that address the same task. Finally Section 6.7 provides guidance to implementing a new task-specific language that addresses a task for which none of the languages above is deemed suitable.

Chapter 7 presents the detailed design of the orchestration mechanism that aims at providing facilities for integrating and coordinating tasks implemented using different languages of the platform. Section 7.1 motivates the design of a coordination mechanism by presenting an exemplar MDE process that involves different MDE and non-MDE tasks. Section 7.2 introduces the ANT tool atop which the orchestration workflow has been designed and the rationale for reusing it instead of designing a solution from scratch. Section 7.3 outlines the challenges that arise from integrating MDE tasks in a general purpose orchestration mechanism such as ANT. Section 7.4 presents the MDE-specific facilities built atop the general infrastructure provided by ANT to address the integration challenges and accommodate the needs of model management tasks. Finally, Section 7.5 presents the individual ANT-tasks that have been implemented for each task-specific language of the platform and Section 7.5.7 outlines the process of contributing a new model management ANT task.

Chapter 8 discusses interesting aspects of the reference implementation such as the choice to build the development tools atop the Eclipse platform and the choice to implement the execution engines of the languages of the platform using a lightweight architecture that utilises interpreters instead of compilers.
Chapter 9 assesses the validity of the hypothesis based on the findings of the research. Section 9.1 presents a case study that served as a common base for evaluating MDE tools in the context of the International Model Driven Development Tools Implementers Forum (MDD-TIF 2007) and its solution using Epsilon. MDD-TIF 2007 was a forum in which, along with York/Epsilon, some of the most active MDE tool-vendors such as Microsoft, MetaCase, INRIA/ATLAS group, openArchitectureWare group, Vanderbilt University and Xactium participated. Section 9.2 evaluates the impact of this work in the MDE community in terms of external users and visibility. Section 9.3 evaluates the benefits in terms of reuse delivered by the proposed approach to specification of model management languages. Section 9.5 discusses the limitations and shortcomings of the proposed approach and Section 9.6 evaluates the current reference implementation.

Chapter 10 concludes by summarizing the findings of this thesis and providing directions to further work in the field of model management language development, and suggestions for transferring the paradigm of integrated task-specific languages to a broader context.

Finally, Appendix A provides a discussion on the graphical notation used in this thesis to illustrate UML models and MOF metamodels, Appendix B provides the complete grammars of the model management languages proposed in Chapters 5 and 6, and Appendix C contains a list of all the acronyms used in this thesis, and their resolutions.
Chapter 2

Literature Review

The aim of this chapter is to provide a critical review of the related work and literature in the field of model management in the context of Model Driven Engineering. The review is separated into five main sections. Section 2.1 discusses Model Driven Engineering and outlines its rationale and objectives. Section 2.2 discusses the characteristics that models should possess in order to be suitable for use in an MDE process. Section 2.3 provides an overview of the most widely-used metamodelling architectures that enable engineers to specify modelling languages and models that conform to them. Section 2.4 discusses the most frequent tasks performed in the context of an MDE process, as well as the benefits and shortcomings of existing tool-support for each task. Finally, Section 2.5 provides a discussion on reuse, interoperability and uniformity among different task-specific languages.

2.1 Model Driven Engineering

Dictionary.com defines a model as: A small object, usually built to scale, that represents in detail another, often larger object or A preliminary work or construction that serves as a plan from which a final product is to be made, or Such a work or construction used in testing or perfecting a final product. Putting the above in the context of engineering, a model is an artefact that represents a real world object - at a level of abstraction - giving the
opportunity to engineers to modify, test and preview certain properties of
the actual object in a cost and effort-effective manner. Model construction
also allows developers to receive feedback and validation of their intentions
by the stakeholders early in the process [42].

There are certain kinds of engineering activities in which models can
be even more useful than as suggested above. For example, in the digital
circuit industry, circuit models are utilized by a combination of computers
and robots in order to produce the actual hardware circuits without human
intervention [43]. This is the essence of Model Driven Engineering: to con-
struct models that describe real-world objects and then derive the designed
objects directly from the models in an automated manner.

In the context of software development, MDE refers to the process of
designing models of software systems and then using automated mechanisms
to derive the actual software from them. MDE promises increased short-
term productivity by speeding up delivery times [44]. To deliver this benefit,
MDE is based on the concept of a highly automated environment for model
management. Moreover, MDD anticipates increased long-term productivity
by reducing the rate at which primary artefacts become obsolete [45]. This
happens because models are expressed using concepts much less bound to
the underlying implementation technology and more closely related to the
problem domain [34].

2.2 Software Models

Models can be generally classified as structured or unstructured according to
the strictness of the structural rules they comply to. Unstructured models
are merely conceptual artefacts that do not conform to rigorously specified
rules, thus giving designers the opportunity to express their views of the
system with unlimited freedom. Examples of unstructured models include
descriptions in natural language, whiteboard diagrams, or drawings in image
processing software. On the other hand, structured models have a well-
deﬁned set of structural elements and rules that they must obey. These rules
can be provided either informally (e.g. with natural language statements)
or in a structured artefact that is called a *metamodel* (also referred to as *modelling language*).

A further classification of structured models can be performed based on the formality of their semantics. Formal models have precise and unambiguous semantics based on well-understood and accepted mathematical concepts while semi-formal models define some of their semantics with informal notations such as natural language to allow different interpretations in different contexts, thus providing greater flexibility. Examples of formal models are those expressed in languages such as Z [38] and B [46] while semi-formal modelling technologies include among others MOF [47], EMF [48] and the newly proposed Microsoft Domain Specific Languages Framework [49].

Unstructured models are particularly challenging to parse and interpret in an automated manner. Therefore, in the context of MDE only structured models are taken into consideration. With regard to the aspect of formality, models for MDE should ideally be rigorously defined. However, formal modelling techniques require a strong mathematical background and are therefore considered to be difficult to apply by mainstream software developers [50]. Moreover, there are a number of issues such as cost-effectiveness, usability and tool support that have driven the current research and practice on MDE to using semi-formal object oriented modelling technologies such as MOF and EMF rather than to using formal notations. In the sequel, the most widely used contemporary object-oriented modelling technologies are discussed.

### 2.3 Metamodelling Architectures

A metamodelling architecture enables engineers to define the abstract syntax of modelling languages and models that conform to those modelling languages. To achieve this, a metamodelling framework typically provides a three-layered (M1-M3) hierarchical architecture as seen in Figure 2.1. M3 contains the core metamodelling language which is used to define modelling languages, M2 contains modelling languages defined using the metamod-
elling language of M3, and M1 contains models conforming to languages contained in the M2. Several metamodelling frameworks have been proposed including the OMG Meta Object Facility (MOF) [47], the Eclipse Modeling Framework (EMF) [48] and, more recently, the Microsoft Domain Specific Languages Tools [49], the Generic Modeling Environment (GME) [51], the MetaEdit+ framework [52], Atom3 [53] and the eXecutable Modeling Framework [54]. The following sections provide a short overview of the most widely used frameworks: MOF, EMF and the Microsoft Domain Specific Languages Tools.

2.3.1 Meta-Object Facility (MOF)

The OMG (Object Management Group) [55], is an organization formed by leading information technology companies, in an effort to standardize component and modelling technologies. The Meta Object Facility (MOF) [47] is an OMG standard that provides a meta-data management framework,
and a set of services to enable the development and interoperability of model and meta-data systems \[47\]. Using MOF, developers can define the abstract syntax of modelling languages, validate models against metamodels and serialize them into an XML format called XML Meta-data Interchange (XMI) \[56\]. MOF is also complemented by the Object Constraint Language (OCL) \[57\], an expression language tailored for defining complex structural constraints that MOF itself cannot capture.

The most well known metamodel defined in MOF and OCL is the UML metamodel that defines the structural rules and semantics of the Unified Modelling Language \[37\], a standardized and industry de facto modelling language for object-oriented systems.

2.3.2 Eclipse Modeling Framework (EMF)

The Eclipse Foundation \[58\] is an organization formed by major software vendors and its main purpose is to provide a platform for development tools interoperability. The Eclipse Modelling Framework (EMF) \[48\] is a modelling framework built atop Eclipse in an effort to provide a practical approach to modelling and model management for Java developers. EMF provides an integrated graphical editor for specifying metamodels, using its ECore M3 language, and a framework (EMF.Edit) for generating graphical editors for new modelling languages from their ECore metamodel. Moreover, EMF provides tools for extracting models from Java annotated source files and XML Schema \[59\] documents. While EMF started as an independent modelling framework, in its latest versions it has been aligned with the MOF 2.0 featuring model validation with OCL and storage capabilities using XMI.

A number of tools have been recently implemented atop EMF including the Graphical Modeling Framework (GMF) \[60\] for building graphical modelling tools, the EMFT Compare tool for comparing different versions of EMF models, and the CDO/Teneo components \[61\] for persisting EMF models in relational databases.
2.3.3 Microsoft DSL Tools

The Microsoft Domain Specific Languages Framework [49] is a framework for implementing tool-support for Domain Specific Languages within the Microsoft Visual Studio development environment. The framework introduces a new metamodeling language that can be used to define the abstract and visual syntax of modeling languages. From a metamodel, tools exist that can generate model editors that can be integrated with Visual Studio. The serialization format of both models and metamodels is, as with the previous two approaches, a dialect of XML.

2.4 Model Management Tasks and Tools

The metamodeling architectures outlined in Section 2.3 enable model engineers to define modeling languages and models that conform to them. To achieve the goals of MDE, defining precise modeling languages and models is not sufficient; a number of tasks also need to be performed on models to enable gradual refinement from disconnected and abstract artefacts into an integrated concrete implementation. The following sections provide an overview of the most frequently-needed model management tasks; namely model validation, model comparison, model transformation, model merging and model-to-text transformation. For each task, a selection of existing tools and languages that enable their automation are also reviewed. The list of model management tasks is not - nor could it be - exhaustive; examples of additional model management tasks that are not discussed in this review include model synchronization, model simulation, text-to-model transformation and impact analysis.

2.4.1 Model Validation

Development of large systems typically involves a number of individuals, contributing to the specification and implementation of different perspectives [62, 63]. Since perspectives often intersect with each other, inconsistencies can be introduced within or among models. Even for legacy systems
that are known to be in a consistent state, there is a significant risk that re-engineering activities (such as refactoring and restructuring) give rise to unexpected inconsistencies [64].

In general, inconsistency can appear in two different forms [65]: incompleteness and contradiction. Incompleteness arises from missing information while contradiction occurs when incompatible information appears in models. An example of incompleteness is the existence of a UML attribute for which no type is declared. On the other hand, an example of contradiction is the existence of two classes which, while not linked with a dependency association, appear to interact in a sequence diagram.

As discussed in [66] maintaining consistency is a time and resource consuming activity. Moreover, in early cycles of the modelling process, inconsistency (and particularly incompleteness) is inherent since some design decisions must be left for later cycles. Therefore, in general, managing inconsistency is not always equivalent to eliminating it. Other options such as ignoring, deferring, ameliorating and circumventing are also available [67, 66].

In development processes that classify models as secondary artefacts, model inconsistency can be tolerated throughout the development process. In MDE however, the key concept is the incremental refinement of models into working systems. Thus, inconsistencies in models are certain to be propagated directly into the implementation making the produced system flawed. Therefore, discovering inconsistencies and ensuring that models are in a consistent state is a crucial requirement for the success of an MDE process.

The following sections provide a further classification of consistency into internal and external, along with a discussion on the characteristics of each type.

**Internal Consistency**

Each model is a self-contained entity and to be internally consistent it must conform to its metamodel, its domain and not contain contradictions.
The following paragraphs discuss those requirements and provide a counterexample for each.

**Metamodel Consistency**  For a model to be consistent, it must at least conform to the language in which it is defined (its metamodel). That means that a model must only contain constructs that the language supports and moreover, these constructs must be interconnected in a manner that satisfies the constraints set by the language.

An example of metamodel inconsistency is the existence of a UML class that inherits itself (since the UML metamodel explicitly defines that such a relationship is illegal [37]).

**Domain Consistency**  Generic modelling languages like the UML have an extensive scope of applicability and allow users to define models of system belonging to diverse domains. However, each domain enforces a set of constraints that a generic language is unaware of. Consequently, there can exist models that comply with the language but not with the domain addressed. Therefore, apart from metamodel consistency, compliance to the domain must also be achieved.

As an example of domain inconsistency we illustrate the case of designing normalized relational database schemas with a generic language such as the UML. In normalized databases (domain), many-to-many relationships between tables are prohibited. Nevertheless, UML has not been tailored for database schema design and is therefore unaware of the specific constraints the domain implies. Thus, while a UML model that represents a normalized database schema and uses many-to-many relationships is valid against the language, it is not semantically consistent with its domain.

**View Consistency**  Finally, even if a model complies with its metamodel and obeys the domain-specific constraints, it can still contain inconsistencies. That is particularly the case in multi-perspective languages [62] in which a model can contain overlapping views of the same artefacts. For example, in UML, the behaviour of a system can be modelled using different types of
diagrams including State Machines and Collaboration Diagrams. Therefore, when depicting a behaviour using more than one of these constructs, there is a possibility of internal contradiction.

**External Consistency**

While internal consistency of models is a necessary foundation, it is not sufficient as models do not exist in isolation. In a typical model driven process, a number of models is involved. Some models can be refinements of other models or depict complementary and possibly overlapping aspects of the system. Therefore, it is essential that models are also maintained in a state where they do not contradict each other. More specifically, two types of external consistency are identified [68, 69, 64]: horizontal and vertical (or evolution) consistency.

**Horizontal Consistency**  In a development process, different but overlapping aspects of the system can be captured in different models. Moreover, with the advent of Domain Specific Languages, models of the same system can be expressed in different modelling languages. While this approach increases the modularity of the system design, it introduces the risk of potential cross-model inconsistencies. Such inconsistencies, if not discovered at an early stage, will be revealed during the integration phase and require significant effort to resolve.

**Vertical Consistency**  Model refinement is a key practice in Model Driven Engineering [21]. Models are initially designed at a high level of abstraction and are incrementally enriched to a level in which they are directly transformable to code. The refinement process is inherently a task that alters the semantics of the refined model. However, when refinement is performed manually, the semantics of the refined models can change undesirably. For instance, concepts or entire use cases of the system may be neglected, behaviour can deviate from the one defined in the early versions of the models that depict the original requirements of the system. Such deviations can
make it difficult to ensure traceability between different versions of models and influence the final implementation of the system negatively.

An important characteristic of this category of consistency issues is that if not detected early in the process they are most probable to be detected during the phases of system or acceptance testing. Therefore, it is important that automated mechanisms exist for evaluating models against their refinements.

**Summary of Model Consistency Classification**

Table 2.1 summarizes the types of model consistency discussed in this section. The first column refers to the type of inconsistency, the second to the phase in the development process in which the inconsistency is more probable to be revealed if it is not managed during the modelling phase and the third depicts the effort that will have to be spent in order to resolve it.

**Model Validation Approaches**

A technique for achieving model consistency is to analyse models by running algorithms that identify inconsistencies [70]. Following that, discovered flaws are corrected mainly manually [71], although work on automatic correction for specific types of models [72] has also been reported.

**Detecting inconsistencies** A directly applicable approach to detecting inconsistencies is to analyse models using a 3rd generation (3GL)(e.g. Java, C++) or a scripting language (e.g. JavaScript). An advantage of this approach is that object oriented APIs exist for almost all types of models

<table>
<thead>
<tr>
<th>Type</th>
<th>How/When Revealed</th>
<th>Resolution Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metamodel consistency</td>
<td>Model validation</td>
<td>Small</td>
</tr>
<tr>
<td>Domain consistency</td>
<td>Unit testing</td>
<td>Medium</td>
</tr>
<tr>
<td>View consistency</td>
<td>Unit testing</td>
<td>Medium</td>
</tr>
<tr>
<td>Horizontal consistency</td>
<td>Integration testing</td>
<td>High</td>
</tr>
<tr>
<td>Vertical consistency</td>
<td>Acceptance testing</td>
<td>High</td>
</tr>
</tbody>
</table>

Table 2.1: Consistency Classification
(e.g. MOF, EMF), and thus it is immediately implementable. However, as discussed in the sequel, inconsistency detection is inherently pattern-based while 3GLs are procedural. Moreover, as demonstrated in Listing 2.1, 3GLs do not provide specialized mechanisms for navigating models and therefore a significant amount of code is required for model navigation. Finally, the consistency checking code depends on the API of the libraries that manage each type of model, and therefore it is challenging to port it to other development platforms.

Listing 2.1: Example constraint in Java

```java
ClassifierClass cc = ...;
Iterator iit = cc.refAllOfType().iterator();
    Classifier c = (Classifier) iit.next();
    boolean satisfied = false;
    Iterator fit = c.getFeature().iterator();
    while (fit.hasNext() && (!satisfied)){
        Feature f = (Feature) fit.next();
        if (f instanceof Attribute){
            if (((Attribute) f).getName().compareTo(c.getName() + "Id") == 0){
                satisfied = true;
            }
        }
    }
if (!satisfied){
    constraintFailed(c);
}
```

Listing 2.2: Example OCL constraint

```oclm
context Classifier
inv MustContainIdAttr: 
self.feature->exists(a:Attribute|a.name = self.name+'Id')
```

A more sophisticated approach is to express consistency rules using higher-order languages that can be either directly executed or used to generate code in a 3GL. Such languages must feature mechanisms for model
navigation using a compound syntax so that rules are maintained brief and expressive. To demonstrate the benefits in terms of clarity and size of specification, in Listing 2.1 a constraint that states that each UML class must contain an attribute named (\texttt{class.name}Id) is expressed in Java, while in Listing 2.2 the same constraint is expressed in OCL.

**Object Constraint Language (OCL)** The Object Constraint Language \cite{57} is an OMG standardized language for specifying constraints on MOF and UML models. In OCL, constraints are expressed as \textit{invariants} in the context of classes. When evaluated, invariants return true or false for each instance of the class depending on whether they hold or not. The concrete syntax of OCL is tailored to support complex model navigation expressions and built-in operations such as \texttt{select()}, \texttt{collect()} and \texttt{iterate()} can be used to query models in a succinct manner.

OCL has been designed to be a side-effect free language. The rationale for this design decision was that it was originally intended to be used only as a constraint language and as such it should preserve the state of the models on which it applies. However, as discussed in a later section, parts of OCL have been used for tasks other than consistency checking.

OCL is inherently suitable for expressing metamodel consistency rules since it was partly created for that specific reason. Examples of metamodels that contain OCL constraints include the UML, MOF and CWM \cite{73}. With regard to view consistency, in \cite{69}, OCL has been used for checking between different diagrams (views) of UML models. Finally, OCL can also check domain consistency since it is used in the definition of UML Profiles. UML Profiles are extensions for expressing concepts that are close to those of UML but that the language does not natively support (e.g. concepts from the database design domain such as \texttt{Table}, \texttt{Column} etc.). With regard to tool support, there are a number of open-source tools (e.g. OCLE \cite{74}, Octopus \cite{75}, Dresden OCL Toolkit \cite{76}) that facilitate evaluation of OCL invariants and error reporting.

While OCL has been used extensively to achieve internal model consistency, the situation is not the same in the context of external consistency.
The reason for this is that an OCL expression (e.g. an invariant) is limited in applicability to the context of a single model.

**Xlinkit** Xlinkit [77] is a mechanism for checking consistency issues in distributed documents. Using Xlinkit, developers can specify cross-document constraints that can be automatically evaluated to reveal inconsistencies. For the specification of constraints, Xlinkit defines an XML-based language that uses XPath [78] for document navigation. In [68] Xlinkit is used as a consistency checking mechanism for UML models, exploiting the fact that UML (and MOF-based in general) models can be serialized into XMI which, as mentioned in Section 2.3.1, is an XML dialect.

Listing 2.3 demonstrates an exemplar Xlinkit constraint that applies on a UML and a Java model and states that for each class in the UML model, a class with the same name must exist in the Java model.

Listing 2.3: Example Xlinkit constraint

```xml
<globalset id="$classes" xpath="//Foundation.Core.Class[@xmi.id]"/>
<globalset id="$javaclasses" xpath="/java/class"/>
<consistencyrule id="r1">
  <description>
   Every class in the UML model must be implemented as a Java class
  </description>
  <forall var="c" in="$classes">
   <exists var="j" in="$javaclasses">
    <equal
     op1="$c/Foundation.Core.ModelElement.name/text()"
     op2="$j/@name"/>
   </exists>
  </forall>
</consistencyrule>
```

While Xlinkit is the only publicly available framework that can be used to automate external consistency checking of MOF-based models, the storage level (XML) is an inappropriately low level for expressing constraints on models. Moreover, the concrete syntax of the expression language is based on XML and that, as illustrated in Listing 2.3, results in lengthy and challenging...
to read and maintain specifications.

**Description Logic (DL)** In [64] the authors propose using Description Logic to check the consistency of UML models. The main motivation for this approach is the lack of OCL tools and the existence of well-tested and working DL tools. This approach is implemented by transforming UML models serialized as XMI into DL specifications using XSLT as a transformation mechanism. DL specifications are then evaluated using the respective tools (*Loom* and *Racer* in this study) to reveal potential inconsistencies. Listing 2.4 depicts a sample constraint on UML models that defines that each `AssociationEnd` has exactly one participant and belongs to exactly one `Association`.

Listing 2.4: Example Loom constraint

```plaintext
(LOOM:defconcept AssociationEnd
 :is (:and ModelElement
 (:at-most 1 Has-participant)
 (:at-least 1 Has-participant)
 (:at-most 1 AssociationEnd-association)
 (:at-least 1 AssociationEnd-association)
 ...)
```

This approach targets only intra-model consistency, a domain which is also covered by OCL. However, it introduces the overhead of transforming UML models into DL specifications. Moreover, the concrete syntax of DL languages such as *Loom*, as presented in Listing 2.4, is challenging to use by contemporary developers with an object-oriented background.

**Transforming to formal notations** There is a significant amount of literature on work that is based on transforming semi-formal models (such as UML) into formal notations such as Z and B [79]. The motivation for this approach is the existence of stable and proven tools that can check such formal specifications in contrast with the shortage of such tools for semi-formal models.

A major concern in this approach is traceability [80]. When using a tool to analyse a formal specification derived by a semi-formal model, the tool
will report potential inconsistencies in the context of the formal specification. However, is not always trivial to locate the source of inconsistency in the original model based on that information.

Another concern about this approach is its applicability in mainstream development processes. Composing and maintaining formal specifications requires a strong mathematical background and extensive training. However, typical contemporary software practitioners lack such skills and doubts have been expressed about the scalability and cost-effectiveness of applying such techniques in mainstream development [50].

Summary

With regard to intra-model consistency, OCL is undoubtedly the most suitable and accepted approach since it is a standard language that has been designed for this specific reason. While OCL has received little criticism on its essence, the main motivation for adopting alternative approaches such as scripting languages, description logic and mapping to formal methods has historically been the lack of stable OCL tools. However, with the advent of improved tool support, there is little motivation for adopting a radically different approach. In the context of inter-model consistency, the Xlinkit toolkit is the only effective, though less than optimal, mechanism presented in the literature (since OCL is limited to expressing constraints on a single model).

In this context, an extension to OCL or a new task-specific language that would enable developers to express and evaluate constraints that involve elements from a number of models, of potentially different metamodels, would be highly valuable.

2.4.2 Model Transformation

Model transformation is a frequently performed task in the context of an MDE process. Generally, two types of model transformations are recognized: mapping transformations and in-place model transformations. As a high-level distinction, a mapping transformation translates a number of
source models into a number of target models which usually conform to a different modelling language while an in-place model transformation performs modifications in the source model itself. The following sections provide an overview of the two types of model transformation and the state-of-the-art tool support that currently exists for automating them.

**Mapping Transformations**

A mapping transformation is an algorithm that defines how a number of source models are mapped to a set of target models \[81\] by encoding mappings of elements of the source into conceptually equivalent elements in the target models. In the related literature, there are numerous references of frameworks that support definition and execution of model transformations including QVT \[82\], ATL \[83\], VIATRA2 \[84\] and Tefkat \[85\]. In this section properties of such frameworks and their variation points are discussed.

As discussed in \[86, 87\], mapping transformations are predominantly rule-based. A rule, in the context of a transformation, is a sub-algorithm that defines how a specific set of elements from the source model should be mapped to an equivalent set of elements in a target model. Each rule can define a domain (the elements of the source model it applies on) and a range (the elements it produces/updates) in the target model. As for the scheduling of rules’ execution, it can be either explicit (users define which rule will be invoked for each element), implicit (the execution engine decides which rule will be invoked for each element) or mixed (implicit, with some user interaction). From a structural perspective, rules can be organized in groups, based on their domain, range or ad-hoc, to enhance modularity and readability. With regard to traceability, each transformation produces a trace that links source elements with their equivalent target elements. Traceability information can be stored either in the source, the target model or in a separate model. Finally a transformation can be either unidirectional or bidirectional. Unidirectional transformations have a predefined direction of applicability while bidirectional can be executed in both directions. With respect to these properties, a high-level classification of the existing
approaches is presented in the sequel.

**Purely Imperative Transformations** In this approach, the source and the target models are subjected to direct, imperative manipulation from an executable language (e.g. Java, XTend [88], Kermeta [89]). Using this approach, particularly complex transformations can be implemented since there are no assumptions or limitations other than those imposed by the programming language and the API of the object models. However, under this approach developers need to implement rule scheduling and traceability from scratch. This introduces an initial overhead and also results into proprietary designs and code that is difficult to reuse, port and maintain.

**Purely Relational/Declarative Transformations** This category contains declarative approaches that have mathematical relations as their base concept. The rationale is to specify the source and target element of a relation using declarative constraints. In their pure form, relational transformations cannot be executed. However, declarative constraints can be assigned executable semantics by mapping to executable languages.

A typical example of such an approach is presented in [90]. In the transformation framework the author proposes, relations between source and target elements are specified in a declarative language (OCL) and are then translated into executable Java code that realizes the transformation. A similar approach presented in [91] also uses OCL for expressing relations but realizes them in XSLT [92].

A characteristic of relational transformations is that they cannot have side effects on the source model. However, as discussed in [93], practical experience with transformations has shown that there are complex transformations for which declarative statements are not sufficient (imperative statements are needed as well) and can therefore not be expressed with this category of transformation approaches.

**Declarative-Imperative Hybrid Transformations** As discussed, imperative approaches make even complex transformations possible but are
difficult to write and maintain. On the other hand, declarative transformations are easy to write and maintain but do not apply to some complex problems. The rationale of hybrid transformation languages is to make simple things simple and complex things possible \[94\] by enhancing declarative approaches with imperative features that can be used where the prior are not sufficient.

A typical example of this category of approaches is ATL \[83\]. ATL is a declarative transformation language in principle but offers native model access features as well to realize complex transformations. In Listing 2.5 a fragment of an ATL transformation is presented. The $\leftarrow$ operator represents a relation while the $\{\}$ block contains an imperative (native) statement on the model.

Listing 2.5: Example ATL Transformation fragment

```plaintext
rule R {
  from
  s : UML!Class
  to
  t : MOF!Class {
    namespace $\leftarrow$ s.namespace
  }
  do {
    t.refSetVal(e, 'name', s.name);
  }
}
```

**In-Place/Update Model Transformations**

Unlike mapping transformations which produce new models, in-place transformations, also known as update transformations, perform modifications of the actual models they apply to. They can be further classified into two subcategories: update transformations in the small and in the large. Update transformations in the large apply to sets of model elements calculated using well-defined rules in a batch manner. On the other hand, update transformations in the small are applied in a user-driven manner on model elements.
that have been explicitly selected by the user.

Some modelling tools provide built-in transformations (wizards) for automating common repetitive tasks. However, according to the architecture of the designed system and the specific problem domain, additional repetitive tasks typically appear, which cannot be addressed by the pre-conceived built-in wizards of a modelling tool.

To address the automation problem in its general case, users must be able to easily define update transformations (wizards) that are tailored to their specific needs. To an extent, this can be achieved via the extensible architecture that state-of-the-art tools often provide and which enables users to add functionality to the tool via scripts or application code using the implementation language of the tool.

Nevertheless, as discussed in [3], the majority of modelling tools have a proprietary API through which they expose an edited model, and therefore such scripts and extensions are not portable to other tools. Moreover, scripting languages and third-generation languages such as Java and C++ are not particularly suitable for model navigation and modification [3].

Existing languages for mapping transformations, such as QVT and ATL, cannot be used as-is for this purpose. The main reason is that such languages have been designed to operate in a batch manner without human involvement in the process. By contrast, as discussed above, update transformations are inherently user-driven.

In [95], an approach to expressing model refactorings as update transformations is described. In this work, update transformations are expressed as guarded actions. The main shortcoming of this approach is that it does not consider user input. This is somewhat limiting as user input is essential for implementing interactive user-driven transformations. Moreover, it is not clear that the language can be used to capture wizards with complex applicability criteria that involve more than one model element, since the examples presented in this work address wizards that apply to a single model element only.

In [96], the authors propose a graphical approach to defining update transformations targeting the Eclipse Modeling Framework. The transfor-
mations can then be interpreted using the AGG graph transformation engine \[97\] or compiled to Java. The transformation language proposed in this tool \[96\] is rather simple and in absence of a proper query language (e.g. OCL), it is not clear how it can be used for complex scenarios.

In \[98\], a set of refactorings on UML models is proposed and specified using OCL pre and post conditions. The paper refers only to specifications and not implementations, and there is no mechanism for executing them to perform the refactorings automatically.

**Summary**

Many languages of different styles have been proposed for mapping transformations. Hybrid languages such as QVT and ATL appear to be most suitable for the task by providing a mixture of declarative constructs which enable developers to easily implement simple transformations, and imperative constructs that can be used when more complicated transformation rules need to be expressed. Moreover, mapping transformation languages have been also shown to be of use for implementing in-place model transformation in the large. By contrast, as discussed above, in-place model transformation in the small is not supported by respective languages and tools to the same degree.

**2.4.3 Model Comparison**

Model comparison is the process of establishing correspondences of interest between elements that belong to different models, that are potentially expressed using different modelling languages and/or technologies. This section discusses the importance of an automated model comparison mechanism as a prerequisite for model integration/merging and model transformation testing and provides an overview of the state-of-the-art approaches to establishing correspondences between elements belonging to different models.
Model Integration

In the relevant literature, the subject of composition of structured artefacts such as models [99, 100, 101], database schemas [102, 103, 104] and XML schemas [105] has been extensively studied. As discussed in all of the above, a prerequisite of composition is the identification of common elements contained in the two so that the merged document does not contain duplicated information. Moreover, to ensure that the merged artefact will not contain contradictions, it is vital to check that the source artefacts conform to each other; i.e., they are composeble.

Model Transformation Testing

While model transformation is a key concept of Model Driven Engineering [81] and a large number of transformation languages have been proposed [106, 107], the aspect of transformation testing is significantly underdeveloped. As discussed in [108], without automated testing it is difficult, especially for large models, to ensure that a transformation is complete and has the intended functionality. Therefore, the authors of [108] identify the need for a generic model comparison mechanism as a fundamental prerequisite for a transformation testing framework. In further work [109], the authors suggest manual construction of the expected outcome of the transformation and comparison with the actual outcome of the transformations using a simple graph-comparison algorithm since the models in comparison (expected and actual) are of the same metamodel.

Model Comparison Approaches

A significant number of techniques and algorithms have been proposed for document comparison with the majority of them applying to textual files [110, 111] or structured documents [112, 113] (e.g. XML files). However, as discussed in [114], although models can eventually be serialized in structured text files, file and tree-based approaches are unsuitable for comparing models since they operate at a significantly low abstraction level.
The majority of approaches that specifically target model comparison generally apply to a single modelling language. For example, the approach presented in [114] targets only UML models. Nevertheless, in the context of Model Driven Engineering and with the advent of approaches based on Domain Specific Languages [49, 52], it is essential that models of different languages and technologies can be compared as well.

In the context of metamodel agnostic approaches, in [115] model comparison is performed with a signature-based approach using MOF unique identifiers as signatures for the respective model elements. This approach exploits the fact that MOF compliant model repositories assign a unique and persistent identifier to each model element upon creation. Therefore, when comparing different versions of the same model, elements can be matched according to their unique identifiers. Still, the applicability of this approach is limited to MOF models that are descendant versions of a common ancestor model.

A metamodel agnostic approach for comparing models of diverse technologies is the Xlinkit approach proposed in [77]. In [68], Xlinkit is used to check the consistency of MOF-based models exploiting the fact that such models are stored as XMI (a dialect of XML) documents. Since other modelling technologies such as EMF [38] and the Microsoft DSL Framework [49] also advocate model serialization in XML, Xlinkit can be even used to compare models of different modelling technologies. Even so, the concrete syntax of the expression language Xlinkit employs is XML-based and this results in lengthy and difficult to read and maintain specifications. Moreover, the storage level (XML) is arguably an inappropriately low abstraction level for expressing constraints on complex models since there are certain model-specific features such as inheritance and details of cross-referencing, the semantics of which are defined in the metamodel rather than in the serialized XML documents.

In [116, 117], the authors propose a signature-based approach to model comparison. Unlike most signature-based methods where the signature of each model element is calculated automatically, in this work it can be defined using an imperative model management language (Kermeta [89]). However,
as the authors mention, this approach applies mainly to structural models as it is quite challenging to calculate signatures for elements belonging to behavioural models (e.g. state machines, sequence diagrams). Also, this approach is limited to comparing models that conform to the same metamodel.

**Summary**

While a number of approaches have been proposed for comparing and establishing correspondences between different models, the vast majority only addresses the case where the involved models conform to the same metamodel. Moreover, the proposed comparison algorithms are predominately identifier or signature-based and as such they cannot be used to implement complex model comparisons.

In this context, to enable developers precisely specify complex model comparison algorithms that can establish correspondences between elements of models that potentially conform to different metamodels, at a high level of abstraction, a task-specific language appears to be highly valuable.

**2.4.4 Model Merging**

Model merging (also referred to as *composition* or *integration*) \[101, 99\] is a special case of model transformation where multiple source models are merged into a target model that is consistent with the input models and does not contain redundant information.

In \[100\] model merging is classified into two major types: horizontal and vertical. As discussed in Section 2.4.1 in MDE a system can be described using multiple models of different languages, thus promoting modularity and separation of concerns \[63\]. However, it is essential that partial models can be integrated into a model that represents the complete system (horizontal integration). Moreover, since multiple developers can be contributing to the specification of a single aspect, it is essential that their contributions are integrable (vertical integration).
Phases of Model Merging

Previous research \cite{101, 102} has demonstrated that model merging can be decomposed into four distinct phases: comparison, conformance checking, merging and reconciliation (or restructuring).

**Comparison Phase** In the comparison phase, correspondences between equivalent elements of the source models are identified, so that such elements are not propagated in duplicate in the merged model. Several approaches to comparison, ranging from manual to fully automatic, have been presented in Section 2.4.3.

**Conformance Checking Phase** In this phase, elements that have been identified as matching in the previous phase are examined for conformance with each other. The purpose of this phase is to identify potential conflicts that would render merging infeasible. The majority of proposed approaches, such as \cite{118}, address conformance checking of models complying with the same metamodel.

**Merging Phase** Several approaches have been proposed for the merging phase. In \cite{101, 119}, graph-based algorithms for merging models of the same metamodel are proposed. In \cite{118}, an interactive process for merging of UML 2.0 models is presented. In \cite{116} a generic algorithm for merging models of the same metamodel on signature-based identified correspondences \cite{117} is proposed.

There are two main weaknesses in those methods. First, they only address the issue of merging models of the same metamodel, and some of them, such as \cite{118, 99}, address only a specific metamodel. Also, merging is typically performed using a generic hard-coded algorithm which cannot be extended or customized by the developers to address context-specific requirements.

Recently, in \cite{120}, the authors have proposed an approach to model merging that is based on generating *merging transformations* expressed in
ATL from weaving models established using matching transformations. Unlike other approaches, this approach can also be used to merge models that conform to different, but similarly-structured metamodels. A shortcoming is that to customize the merging behaviour, developers need to modify the generated ATL merging transformations which are particularly complicated due to the fact that ATL has not been designed as a merging language and reasonably it lacks the proper abstractions that a task-specific language would provide.

**Reconciliation and Restructuring Phase** After the merging phase, the target model may contain inconsistencies that need fixing. In the final step of the process, such inconsistencies are removed and the model is polished to acquire its final form. Although the need for a reconciliation phase is discussed in [102, 119], in the related literature the subject is not explicitly targeted.

**Summary**

The existing literature and tooling in the context of model merging is predominantly focused on merging models that conform to a common metamodel. As such, the majority of proposed approaches involves a fixed matching process that is based on identities or signatures and a fixed merging algorithm that merges the input models on the identified correspondences.

As discussed in Section 2.4.2 in the related field of model transformation, developers are provided with powerful task-specific languages that enable them to precisely specify transformation scenarios between models of the same or different metamodel, at a high level of abstraction. By contrast, in the field of model merging developers are only provided with tools that implement fixed matching and merging algorithms and which predominately target models of the same metamodel.

In this context, and given the proven value of task-specific languages for tasks such as model validation and transformation, a task-specific language that enables developers to precisely specify algorithms for merging
models, that potentially conform to different metamodels, at a high-level of abstraction appears to be an important and valuable facility that is currently missing.

2.4.5 Model-to-Text Transformation

The majority of artefacts involved in a software development process such as code, documentation and configuration files, are in a textual form. Therefore, an essential part of an MDE process is to be able to transform models into textual artefacts that can then be used from classical software engineering tools such as compilers, configuration and help systems.

The field of model-to-text transformation is well-studied and a number of mature languages and tools already exist and are widely used. Among them the most widely-used are the template-based XPand [121], MOFScript [122] and JET [123] languages and the grammar-based TCS [124] language that also provides text-to-model facilities.

During the review, no major research challenges have been identified with regard to existing model-to-text transformation languages, apart from interoperability and consistency with other task-specific languages, an issue which is discussed in detail in the following section.

2.5 Interoperability, Reuse and Consistency of Model Management Languages

The review performed in Section 2.4 identified a set of distinct model management tasks and presented a number of task-specific languages that have been proposed for automating the most recurring of them; particularly model validation, model-to-model transformation and model-to-text transformation. As an MDE process typically involves a number of interdependent model management tasks, the presence of many different languages that can be used to automate different tasks of the same process inevitably raises the issues of integration, interoperability, consistency and reuse across them.
This section provides a review of the interoperability and integration capabilities of existing task-specific model management languages, and a discussion on the reusability of code across different languages.

2.5.1 Integration and Interoperability

In the context of an MDE process several model management and classical development (e.g. code compilation, deployment) tasks need to be performed in a chained manner. For example, to generate a data-intensive application from an input UML model the following steps are required:

1. Validate the UML model
2. Generate Java code from the UML model
3. Transform the UML model into a Database model
4. Generate SQL code from the Database model
5. Compile the generated Java code
6. Deploy the SQL code into a DBMS

The first step of this simple scenario involves validating the source UML model in order to identify inconsistencies that will impact the remaining steps of the process. If the UML model is found to be consistent, the next step is to generate Java code from it using a model-to-text transformation. Following that, SQL code for persisting/retrieving the data managed by the application needs to be generated from the UML model. This is typically achieved in two steps to reduce coupling and promote modularity: in step 3, the UML model is transformed into a model that conforms to a Domain Specific Language tailored to describing Database structures and in step 4 this intermediate Database model is transformed into SQL code using a model-to-text transformation. Finally, the generated Java code needs to be compiled and the SQL code needs to be deployed in the underlying Database Management System (DBMS).
To implement this scenario, three model management languages are required: a validation language such as OCL for step 1, a model-to-text transformation language such as MOF2Text or MOFScript for steps 2 and 4, and a model-to-model transformation language, such as ATL or QVT, for step 3. Also, as the tasks are interdependent, the model-to-text task in step 2 and the model-to-model task in step 3 receive the validated UML model as input, and the model-to-text task in step 4 receive the transformed Database model as input. A graphical overview of the task dependencies appears in Figure 2.2.

The simplest way in which two model management languages can interoperate is by being able to access the same type of models. However, in this case, in every step of the workflow, each language runtime needs to load the models from their physical location and store them (if necessary) afterwards. Thus, if $M_v$, $M_t$, $M_g$ are (respectively) languages for validation, model-to-text and model-to-model transformation, and if they can only be
integrated by accessing the same type of models, the process above would involve the following steps:

1. Load the UML model in \( M_v \)
2. Validate the UML model
3. Load the UML model in \( M_t \)
4. Transform the UML model
5. Store the DB model
6. Load the UML model in \( M_g \)
7. Generate the Java code
8. Load the DB model in \( M_g \)
9. Generate the SQL code

On the other hand, if \( M_v, M_t \) and \( M_g \) shared a common internal model representation scheme, the process could be simplified to the following:

1. Load the UML model
2. Validate the UML model
3. Transform the UML model into an in-memory DB model
4. Generate the Java code
5. Generate the SQL code from the in-memory DB model
6. (optionally) Store the DB model

Loading and storing large models is a resource- and time-consuming activity, and even this simple example demonstrates that using languages that interoperate only at the model format level introduces a number of (potentially expensive) model loading/storing steps which would otherwise be unnecessary.
2.5.2 Code Consistency and Reuse

Another significant issue when using many languages to implement the steps of a workflow is reusability and consistency of code. Consider, once again, a workflow which involves validation, transformation, and model-to-text generation tasks, as above. When managing UML 2.x models, a typical activity in all three tasks is to examine model elements to check if they have certain stereotypes attached. If the three languages selected for these tasks were OCL, ATL and MOFScript, respectively, then the user would need to specify the same helper operation (which checks whether a specified stereotype is attached) three times, using similar but inconsistent syntaxes, as illustrated in Listings 2.6, 2.7 and 2.8 respectively.

```
Listing 2.6: The hasStereotype() helper expressed in ATL

helper context UML2!Element def : hasStereotype(s : String)
    : Boolean
    self.getAppliedStereotypes()->exists(st | st.name = s);
```

```
Listing 2.7: The hasStereotype() helper expressed in OCL

package uml
  context Element
  def Operations:
    let hasStereotype(s : String) : Boolean =
      getAppliedStereotypes()->exists(st | st.name = s)
endpackage
```

```
Listing 2.8: The hasStereotype() helper expressed in MOFScript

uml.Element::hasStereotype(s : String): Boolean {
  result = self.getAppliedStereotypes()->exists(st | st.name = s);
}
```

Duplication of code in different languages introduces a maintenance problem and is also a potential source of coding errors as it requires users to work concurrently with similar but inconsistent syntaxes.
2.5.3 Interoperability, Reuse and Consistency in Contemporary Model Management Languages

Currently the majority of contemporary model management languages (such as OCL, ATL, MOFScript, XTend, XPand, Check, Tefkat etc) support management of models specified atop the Eclipse Modeling Framework. A limited number of languages (such as OCL, ATL and MoTMoT [125]) also support management of MOF 1.4 models using the infrastructure provided by the NetBeans Metadata Repository (MDR). However, in the absence of a commonly accepted abstraction layer, each language runtime provides a custom internal model representation scheme that renders direct interoperation between runtimes of different languages, at least, challenging.

In terms of reuse and consistency, each model management language provides a custom syntax and as a result code written in one language cannot be reused from within another language. Although many model management languages build on OCL, each language typically embeds a subset of OCL that, as shown in Listings 2.6 - 2.8 is not reusable across languages and tools.

2.5.4 Platforms of Model Management Languages

Exceptions to the landscape of disconnected and inconsistent model management languages are the OpenArchitectureWare (oAW) [88] framework, the family of model management languages that are managed by the OMG (OCL, QVT and MOF2Text) and the AMMA platform.

The OpenArchitectureWare (OAW) Framework

The oAW framework [88] provides a set of interoperable languages for model validation (Check), model-to-text transformation (XPand) and a general-purpose imperative model management language (XTend) that are consistent and interoperable with each other.
The AMMA platform

The AMMA platform provides a hybrid language for model transformation (ATL) as well as a language for specifying textual concrete syntaxes for modelling languages (TCS). Instead of providing a different language for each model management task, AMMA follows an approach based on Higher-Order Transformations (HOTs). Under this approach, purpose-specific ATL transformations are generated from higher-order models. As an example of this approach, in [120], model merging is performed by transforming weaving models into ATL transformations that implement the merging algorithm.

OMG Model Management Languages

The OMG has standardized OCL, QVT and MOF2Text that target the tasks of model validation, model-to-model transformation and model-to-text transformation respectively. By design, the languages are consistent with each other as QVT reuses OCL and MOF2Text builds on QVT and OCL.

A detailed discussion on the scope and the organization of these platforms and their relationship to the platform proposed in this work is provided in Section 9.4.

2.6 Chapter Summary

This chapter has presented the core principles of Model Driven Engineering and identified the most frequent tasks that make up an MDE process. For each task, the state-of-the-art tools and languages have been identified and a discussion on their advantages and shortcomings has been provided. Following that, a discussion on integration, reuse and interoperability between the different task-specific languages was presented. This review has identified two main research problems: while tasks such as model validation, model-to-model transformation and model-to-text transformation are supported by task-specific languages, other tasks such as inter-model consistency checking, model comparison, merging and in-place model transformation do not have the same level of support. Also, as task-specific languages have been
predominantly designed in isolation, significant consistency, reuse and inter-operability problems appear when combining them in the context of complex workflows.
Chapter 3

Analysis and Hypothesis

Throughout the field review in Chapter 2, a number of open issues requiring further research have been identified. In this chapter, these findings are summarized and further analysed, the research hypothesis and objectives of this thesis are established and the research methodology is outlined.

3.1 Research Challenges

Study on approaches to model management in MDE has revealed a number of distinct tasks such as model comparison, (inter and intra-model) validation, model to model transformation, model to text transformation, and model merging.

For the tasks of inter-model validation, model comparison and model merging, no task-specific languages of a proper abstraction level have previously been proposed in the literature.

For the tasks of intra-model validation, model to model transformation and model to text transformation, the most successful approaches employ task-specific languages. For example, OCL is used for checking intra-model consistency. Several languages such as QVT [106], ATL [83], VIATRA [84] and Tefkat [85] have been proposed for model transformation. Model to text transformation is also supported by languages tailored to the needs of the task such as MOFScript [122] and XPand [121].
The main reason for the existence of this plethora of languages is that each model management task demonstrates a set of unique requirements. This, in most cases constitutes a task-specific language that natively captures frequent patterns and operations involved in the task more suitable than a general purpose language in which developers have to implement this logic explicitly [3].

However, the existence of many task-specific languages that have been developed independently of each other has introduced a number of challenges in terms of interoperability, integration, consistency and re-use. While each task-specific language has different goals and syntax that reflect the individual requirements of the task it supports, they all share a common set of characteristics. For example, all task-specific languages need to access, navigate and modify models (often more than one simultaneously), define and invoke methods/operations, declare and use local variables and support program control flow constructs (such as if branches and for and while loops).

The majority of contemporary model management languages use a subset of the Object Constraint Language [57] for model navigation. As discussed, OCL provides a powerful syntax for navigating through model elements and collections. Moreover, it has a substantial user basis and this reduces the effort required to learn and apply languages that re-use it significantly.

However, the supported OCL subsets vary among different languages and consequently, OCL expressions that are valid in one language (e.g. a transformation language) may not be valid in another (e.g. a model to text language). This requires multiplication of effort to express identical model queries in multiple dialects of OCL and the existence of many similar copies of the same artefact is a well-known maintenance problem.

Moreover, OCL does not support a number of common features required by model management languages and therefore, in practice, language designers have to implement them from scratch for each new task-specific language. In detail, missing features of OCL when considered as a base-language for development of task-specific model management languages include:
**Single Model Access:** OCL expressions cannot access more than one model at a time. For the task of intra-model validation, which OCL was primarily designed for, this feature is not required. However, for tasks such as inter-model validation or model transformation it is essential to be able to access multiple models simultaneously.

**Read-Only Access:** Another feature absent from OCL is the ability to modify the state of a model. This is a core design decision of OCL as it was originally designed only as a constraint language. However, a vast number of model management tasks such as model transformation and model refactoring inherently need to support model modification capabilities.

**Deep Nesting of Expressions and Statements:** A further limitation of OCL is that it does not support statement sequencing and grouping. While this is tolerable for expressing simple constraints, when constructing more complex expressions, the result is generally deeply nested statements that can be difficult to read and maintain. Finally, in the context of usability, OCL lacks common programming concepts such as for and while loops.

**Low Modularity and Reusability:** Modularity and re-usability are essential features for any programming language. In the context of model management, it is common practice to abstract complex model queries into re-usable operations. For example, when navigating UML models, it is useful to abstract the significantly longer query looking for a particular stereotype in the context of a model element (as displayed in Listing 3.1) with a shorthand `isStereotyped(stereotypeName : String)` operation.

Listing 3.1: Example OCL Query

```ocl
class ModelElement

isStereotyped(stereotypeName : String)

modelElement.stereotype->exists(
    stereotype:Stereotype|stereotype.name = stereotypeName)
```

While in OCL such operations on model elements can be defined (OCL calls them helpers [57]), they cannot be grouped in external re-usable libraries.
**User Interaction:** In a number of occasions, during the execution of a model management program, there is a need to interact with the user to resolve issues that the program cannot decide automatically. Typical such cases are when users need to provide new information (e.g. the name of a newly created UML class) or choose from a list of alternatives. Currently, OCL completely lacks such mechanisms, even for the simplest form of user interaction.

As a consequence, the majority of existing languages such as ATL [83], MOFScript [122], YATL [93] have implemented the features discussed - each to a different extent - from scratch. As a result, users need to learn and remember several languages that are not consistent with each other. Also, since the languages are not compatible with each other, when the same functionality is required by two different programs that address different management tasks, users need to implement and maintain two copies in two different languages. Finally, in the absence of an infrastructure that provides the common features discussed above, implementing a new task-specific language is significantly challenging and expensive and the resulting language and supporting tools typically suffer from the well-known problems of new software, particularly logical defects and incompleteness.

3.2 Research Hypothesis

With respect to the current situation discussed above, the context of the research hypothesis is as follows:

A Model Driven Engineering process can involve many different model management tasks such as validation, transformation (including model-to-model and model-to-text), in-place model transformation, comparison and merging. Currently, there are a number of independently developed task-specific languages that support some of these tasks, particularly model validation, model-to-model and model-to-text transformation. By contrast, no task-specific languages have been proposed for tasks such as model comparison, merging, inter-model validation and in-place transformation. Also, the
fact that existing task-specific languages have been predominantly developed independently of each other leads to consistency, reuse and interoperability problems.

From the language engineering point of view, when constructing a new language in the absence of an underlying framework that provides a set of reusable common facilities, engineers need to reimplement a significant amount of functionality, and experience has demonstrated that this is typically done in an inconsistent manner that results in uniformity and interoperability problems between the new language and existing languages for other model management tasks. From the user’s point of view, the user needs to learn and work concurrently with a number of inconsistent and isolated languages that do not interoperate with each other, and across which code cannot be reused and therefore needs to be duplicated and maintained separately.

In this context, the hypothesis of this thesis is stated as follows:

*Despite their individual requirements and characteristics, a wide range of task-specific modelling languages share a significant number of common features, and therefore, instead of developing each language separately, it is beneficial in terms of reuse, uniformity and interoperability, both from a language engineering and a user perspective, to develop them atop a platform that provides a reusable set of commonly required features.*

The objectives of the thesis are:

1. To develop a platform atop which uniform, interoperable and reusable languages can be developed.

2. To use the platform to develop task-specific languages that address the, now largely unsupported, tasks of inter-model consistency checking, model comparison, model merging and in-place model transformation.

3. To use the platform to develop uniform task-specific languages for tasks that are already supported by existing languages (e.g. model-to-
model transformation, model validation).

4. To develop an orchestration and coordination framework that enables composition of individual model management tasks implemented using languages of the platform into coherent workflows.

### 3.3 Research Scope

The purpose of this section is to establish the scope and boundaries of this work. As discussed in Section 2.4, there is a plethora of model management tasks that can benefit from the automation delivered by task-specific languages. The list of such tasks includes - apart from the aforementioned tasks of model transformation, validation and merging - tasks such as model simulation, synchronization, and text-to-model transformation. Due to the high number of such tasks and time and resource constraints of this work, a decision has been made to limit the scope of this work to the most recurring tasks, and provide guidelines on how task-specific languages for additional tasks can be constructed on top of the infrastructure provided by Epsilon. It is also important to clarify that this work addresses only the technical space of models, and as such it work does not address tasks such as text-to-model transformation.

### 3.4 Research Methodology

To evaluate the validity of the hypothesis, a typical software engineering process involving an initial *analysis* phase, followed by several *design*, *implementation* and *testing* iterations has been followed. A graphical overview of the process appears in Figure 3.1. This section provides an overview of the main phases of the process.

#### 3.4.1 Analysis

In the analysis phase, the most widely used metamodelling architectures that enable engineers to define modelling languages and models, and the
the most common model management tasks were identified. For each model management task, the state-of-the-art approaches in terms of supporting languages and tools were reviewed in order to identify their advantages and shortcomings. Through the analysis, a number of research challenges have been identified that have motivated the hypothesis and objectives of the thesis.

3.4.2 Iterative Design and Implementation

Following the analysis phase, a conceptual architecture has been conceived to investigate the hypothesis. This conceptual architecture has been refined into a technical design which in turn has guided the construction of a reference implementation.

In the first phase of design, infrastructure was designed based on the findings of the analysis. Following this, languages for previously largely unsupported tasks (model comparison and merging) were built atop the infrastructure. Through the process of building these first task-specific languages, the design and implementation of the infrastructure have been progressively improved and become more flexible in order to accommodate the needs of
the different tasks. Latterly, languages for model-to-model transformation, model validation and in-place model transformation were designed and implemented.

Finally, a workflow mechanism was designed and a prototype was implemented to enable orchestration and coordination of tasks implemented using different task-specific language of the platform.

3.4.3 Iterative Testing

Throughout the design and implementation iterations, several case studies have been used to assess the quality and usefulness of the proposed approach and the correctness of the reference implementation. Also significant feedback has been provided by academic peers who have reviewed a number of publications on several core aspects of the design of the platform and the languages that have been built atop it, and by external users who have been using the reference implementation. Errors and omissions identified during testing iterations were provided as input to further design and implementation cycles.

3.5 Chapter Summary

This chapter has provided a detailed discussion on the research challenges identified during the background review performed in Chapter 2 and established the hypothesis and objectives of the thesis. The following chapters discuss the establishment and development of a platform of integrated languages for model management that is used as a means for exploring the proposed hypothesis.
Chapter 4

Architecture

This chapter presents an overview of the architecture of a platform that facilitates the construction of interoperable, reusable, and uniform model management languages, as envisioned in Chapter 3. This platform is hereafter referred to as *Epsilon*, standing for Extensible Platform for Specification of Integrated Languages for Model Management.

Describing the architecture also involves making and presenting several high level decisions concerning the organization and structure of the system which will later guide the design and implementation phases. Moreover, it involves identifying the desired quality attributes the architecture should exhibit as well as the mechanisms through which those qualities are achieved (tactics).

The chapter is organized as follows. In Section 4.1, the desired quality attributes of the architecture are discussed. In Section 4.2, the main architectural tactics employed for achieving the identified quality attributes are presented. Then, Section 4.3 provides a high-level overview of the architecture and Sections 4.4 - 4.8 provide more details about the individual layers of the architecture and their purpose.
4.1 Quality Attributes

Quality attributes are the desired characteristics of the system [126]. They can be divided into two major groups: system and business qualities. System qualities refer to the structural and operational aspects of the system and according to the visibility from a user perspective, they can be further separated into observable and unobservable attributes [126]. Observable system qualities include among others usability, performance, availability, testability while unobservable attributes include modifiability, portability etc. Business qualities refer to characteristics of the system that are relevant to the business context in which it is developed. Therefore, this category includes attributes such as total cost and time-to-market.

It is important to identify the quality attributes early in the architectural process as they will drive the organization and the structure of the system. In this section, the desirable attributes are identified and classified according to their importance in the context of the system.

4.1.1 System quality attributes

Modifiability/Extensibility

The main targets against which platform is designed are extensibility and modifiability. The motivation for this is that little work has been previously reported on designing platforms of integrated programming languages similar to the one proposed in the context of this thesis. Therefore, it is anticipated that the construction of diverse task-specific languages atop the platform, will likely introduce changes in the platform itself so that it can accommodate similar requirements of different task-specific languages in a consistent, uniform and integrated manner.

Usability

The primary aim of providing task-specific languages for different model management tasks is to make it easier for the users to automate specific tasks by using a dedicated focused syntax and a supporting runtime that
automates the uninteresting and error-prone parts of each individual task. In this sense, usability is inherently a primary concern of the proposed approach. Also to obtain feedback from external users, as part of the testing and evaluation methodology as discussed in Section 3.4, the architecture must provide development tools, that enable users to easily compose and execute model management programs.

Performance

In the context of this thesis, the performance of the platform and the languages built atop it is not a primary concern - particularly if it is to be achieved by compromising extensibility or modifiability. However, to be usable, model management programs should generally run in a reasonable amount of time for small and medium-sized models.

4.1.2 Business quality attributes

Total cost

Although this is not a commercial project, its budget can be counted in terms of time. In this sense, and since this is a project with many possible extensions, the total design and development time should be kept within balance with the available time and resources.

4.2 Architectural Tactics

Tactics are mechanisms that are used in order to achieve the desired quality attributes. As mentioned in [126], a tactic is a design decision that influences the control of a quality attribute response. In this section, the tactics used to achieve the quality attributes identified in the previous section are presented.

4.2.1 Maintenance of semantic coherence

Maintaining semantic coherence among the modules of a layer allows the localization of changes and prevention of ripple effects. Therefore, when a
change is required, the scope of the change is clearer and narrower. Moreover, when there are expected changes, assigning clear responsibilities to modules and packages will make it easier to perform the alterations without violating the overall architectural decisions and with minimum restructuring and redesign.

4.2.2 Separate interface from implementation

Separating interface from the implementation allows user interface developers and designers to modify the interface in order to achieve enhanced usability without having to propagate the changes into the implementation layer. It is obvious that this tactic enhances modifiability as well.

4.2.3 Information hiding

Limiting visibility to the minimum possible level reduces coupling and enhances modularity. Moreover, it allows for substitution of components with others that provide the same interface.

4.2.4 Deferring binding time

Use of configuration files delays the binding time from compile to execution and allows the user to extend the behaviour of the system without intervening with the source code. Moreover, use of interfaces allows component replacement without modifications to the rest of the source code.

Based on the identified quality attributes and architectural tactics, the remaining sections present an architectural overview of the platform.

4.3 Architectural Overview

This section provides a high-level overview of the layers of the proposed architecture and discusses the role of each layer in the overall architecture as well as its relationships to other layers. Chapters 5-7 refine this abstract
architecture into a detailed design that contains sufficient detail to construct a prototype implementation of the platform.

The basis of the architecture of Epsilon, displayed in Figure 4.1, is the Model Connectivity layer. This layer enables languages of the platform to access models of diverse technologies and metamodels in a uniform manner. Atop it, the Core Language layer provides a common set of features needed by task-specific model management languages. On top of the Core Language layer, the Task Specific Languages Layer contains a number of task-specific languages that address the tasks of intra and inter-model validation, model comparison, model merging, in-place and model-to-model transformation and model-to-text transformation. The Workflow layer provides support for assembling and coordinating complex model management workflows, and the Development Tools layer contains the tooling through which end-users can specify, debug and execute model management operations. In the following sections, the role of each individual layer is further discussed.

4.4 The Model Connectivity Layer

As discussed in Section 2.3, many metamodeling technologies are currently being used to define modelling languages and models. As Epsilon should not be restricted to a particular metamodeling technology or metamodel, a key purpose of the Epsilon Model Connectivity (EMC) layer is to provide a set of facilities through which model management programs will be able to uniformly access and modify models of diverse technologies. Such facilities include:

**Loading and persisting models** Models are typically persisted in the file-system, databases or dedicated repositories. EMC is responsible for providing a uniform interface through which technology-specific implementations can define the loading and persistence mechanisms that a particular technology employs.
Type-related operations During a model management operation it is often necessary to retrieve all the model elements of a particular type or to retrieve the type of a given model element. To abstract away from the diverse type-system implementations in different technologies EMC is responsible for providing a uniform interface for performing type-related operations.

Obtaining and setting values of model element properties Similarly, modelling technologies provide different implementations for obtaining and setting the values of properties/features (i.e. attributes, references) of model elements. To enable the overlying languages to interact with models of diverse technologies uniformly, EMC provides a uniform interface for
retrieving and modifying values of properties of model elements.

**Transactions** In case a model management operation fails for some reason (such as an unanticipated exception), there is a risk that models it manages are left in an inconsistent state. To address this issue modelling technologies are increasingly providing support for atomic transactions that can be committed or rolled back on demand - similarly to transactions in relational databases. While not all modelling technologies provide such features yet, EMC provides an interface for managing transactions from languages that use it.

The detailed design of the Model Connectivity layer is presented in Section 5.1.

### 4.5 The Core Language Layer

The purpose of this layer is to provide an extensible and reusable core language, hereafter referred to as the Epsilon Object Language (EOL), which accommodates common facilities of task-specific model management languages. Such facilities include:

**Model navigation and querying** This layer is responsible for providing a uniform mechanism for querying and navigating models.

**Model modification** A number of model management tasks such as model transformation, refactoring and merging, predominantly update models. To avoid diverse implementations of model modification features in different task-specific languages, EOL provides a uniform set of such features that can be reused by higher order languages.

**Access to multiple models** Since the majority of model management tasks needs to access more than one model simultaneously, EOL provides mechanisms that support this functionality.
**Statement sequencing and grouping** Sequencing and grouping statements allow developers to disentangle complicated, nested queries making them easier to read and debug.

**Reusable operations** To enable reuse between different model management programs, EOL is responsible for providing support for reusable libraries of operations which can be invoked by any language of the platform.

**Transactions** As the Model Connectivity layer provides support for transactions, the task-specific languages should also be able to manage transactions programmatically. EOL provides the necessary constructs for this purpose.

**Bridge with native code** In some cases, the user of the object language (or a language built atop it) may need to perform an activity which is clearly outside the scope of model management (e.g. to connect to a database server) or a task at which the language is not very efficient at (e.g. to perform complex mathematical computations). To perform such tasks EOL provides support for invoking code written in the programming language in which EOL itself is defined (e.g. Java). This tactic is typical in language construction to avoid dead ends without duplicating functionality from a lower-level language. For example, Java can invoke C code, using JNI [127], and C can invoke Assembly code.

The detailed design of the Core Language layer is presented in Section 5.2.

### 4.6 The Task-Specific Languages Layer

The Epsilon Object Language layer summarised above has been introduced to enable definition of a number of languages, each addressing a specific model management task with little overlap between each other. Each task-specific language in this layer defines its own syntax which however reuses the syntax and facilities provided by EOL. This section briefly discusses the aim and scope of each language in the layer.
Epsilon Transformation Language (ETL)  ETL enables users to specify algorithms that can transform a set of input models to a set of output models of potentially different metamodels.

Epsilon Comparison Language (ECL)  The purpose of this language is to enable users to specify algorithms that compare elements of two models to establish correspondences between them.

Epsilon Merging Language (EML)  The aim of EML is to enable users to specify how two models should be merged into a third one given a set of correspondences (calculated using ECL or otherwise).

Epsilon Validation Language (EVL)  EVL provides the means to express constraints over model elements which, when evaluated, can reveal inconsistencies in or between models.

Epsilon Wizard Language (EWL)  While ETL targets the problem of expressing batch transformations that transform entire models to other metamodels, EWL enables users to specify interactive in-place model transformations.

Epsilon Generation Language (EGL)  The purpose of EGL is to enable users to generate textual artefacts such as code and documentation from models.

The detailed design of the Task Specific Languages layer is presented in Chapter 6.

4.7 The Workflow Layer

As already discussed, in many occasions, a model management process involves more than one distinct step. For instance, to generate SQL code from a UML model the process would involve validating the UML model, transforming it into a Database model that is more closely related to the target
domain using a model-to-model transformation and then generating the SQL code using a model-to-text transformation. The Workflow layer provides support for assembling and executing such model management workflows from individual tasks implemented with languages of the Epsilon platform. The workflow layer provides the following facilities:

**Common Model Repository**   As discussed in Section 2.5 in the context of a workflow, more than one model management task can be performed on the same models sequentially. Moreover, loading and storing models is a resource and time consuming process, particularly as models grow in size. The aim of the common model repository is to provide centralized management of model loading/storing so that individual model management tasks do not need to load/store the models they operate on themselves. This facility contributes both to decoupling and modularity, and to performance as each model is loaded/stored at most once although it may be used by multiple tasks.

**Inter-Task Communication Facility**   This facility enables model management programs expressed in different languages of the platform to communicate with each other at runtime by providing a mechanism that allows individual tasks to export variables (for example containing results of complex queries) so that subsequent tasks can reuse them instead of recalculating them.

The detailed design of the Task Specific Languages layer is presented in Chapter 7.

### 4.8 The Development Tools Layer

This layer provides end-user tools for composing, debugging, executing and monitoring individual model management tasks as well as workflows. Provision of user-friendly development facilities is important as it enables external users to experiment with the languages and other facilities of the platform and provide feedback that will contribute to the evolution of the platform.
To achieve uniformity and interoperability with other tools involved in a typical software development process such as compilers, version control systems and graphical editors, the user-interface should be built atop an extensible integrated development environment (IDE) such as Eclipse [58]. Building the user interface on such an environment should also deliver productivity and quality benefits as a significant amount of stable and tested functionality, such as support for file management, version control and sophisticated source code editing, can be largely reused.

A detailed discussion on the implementation of the Development Tools layer is presented in Chapter 8.

4.9 Architectural Control and External Reuse

As with any software system, during the process of establishing the architecture of Epsilon, two conflicting driving forces appear: architectural control and reuse of external components. Architectural control refers to the degree of control the designer has over key architectural decisions of the system. Reusing external components delivers significant productivity and quality benefits, but also reduces the level of architectural control the designer has on the system as it can directly or indirectly enforce architectural decisions made in the external components, to the system that reuses them. Therefore, during the establishment of the architecture of Epsilon, it is essential to identify the parts of the architecture in which a high degree of architectural control is important and those in which a reduced degree of control can be tolerated, and make an informed and balanced decision over the degree and form of external reuse in each layer.

4.9.1 The Core Language and Task-Specific Languages Layers

As the main focus of this work is on engineering languages for model management, it is deemed essential that a high degree of architectural control is established and maintained over the Core Language and Task-Specific
Languages layers.

Therefore, for the Core Language and Task-Specific Languages of the architecture the decision not to reuse directly external components such as OCL, QVT, AMMA or oAW has been made. While those systems contain different subsets of similar features, none of them contains all the desirable features discussed in Section 3.1, which have been identified as important for a platform of integrated and interoperable model management languages.

While direct reuse in the context of those two core-layers has been deemed architecturally restrictive, some form of indirect reuse may still be possible. Two alternative indirect reuse strategies have been identified: branching and conceptual reuse. In the first strategy, one of the existing components can be chosen as a basis and then be iteratively modified to fit the requirements of the system at stake thus forming a parallel development branch. The main concern in a branching strategy is that although in principle the base component can be modified, an established basis indirectly enforces the architectural style of the resulting system. As this is a research-oriented project with the aim of producing novel ideas on the engineering and capabilities of model management languages, a green-field approach that does not pose any upfront constraints and allows experimenting with different architectural and design decisions is deemed highly valuable.

In the conceptual reuse strategy, reuse is achieved in a soft manner by re-implementing valuable features of the external system in the context of the target system. Given the number of existing languages, there are undoubtedly many proven features that languages of Epsilon should also demonstrate. A conceptual reuse strategy will allow Epsilon to harvest such interesting and valuable features without compromising architectural control. On the other hand, such a strategy shall incur an increased effort for the development and maintenance of the implementation of the system. Taking into consideration the criticality of the two layers for the purposes of this work, the decision to adopt conceptual reuse, and thus accept the increased development effort, has been made.
4.9.2 The Model Connectivity, Development Tools and Workflow Layers

While the strategy of conceptual reuse has been selected for the core layers of the architecture, a different decision has been made for non-core layers. For example, since the scope of this work does not encompass specifying a new model representation technology, the Model Connectivity layer has been designed so that it facilitates extensive reuse of existing metamodelling technologies such as EMF and MDR. Similarly, since it is not in the intentions of this work to construct a new IDE, in the architectural description of the Development Tools Layer it is proposed that the development tools that support the languages of Epsilon are built atop an existing IDE such as Eclipse. Finally, it is anticipated that the Workflow layer can also benefit from reusing an existing component with similar functionality such as ANT or Maven.

4.10 Chapter Summary

This chapter has presented an overview of the architecture of the Epsilon platform that enables development of integrated task-specific languages for model management. It has also discussed the driving quality attributes of the architecture and the architectural tactics that have been employed to achieve them.

The platform consists of five main layers: the Model Connectivity Layer that enables uniform management of models that are captured in different modelling technologies and metamodels, the Core Language that provides a set of uniform features that are meant to be reused by task-specific languages, the Task-Specific Languages layer that provides a number of languages each of which addresses a distinct model management tasks, the Workflow Layer that provides facilities for integrating individual model management tasks implemented using languages of the platform into coherent workflows and the Development Tools layer that provides end-user tools for composing, executing, debugging and monitoring individual model manage-
ment tasks and workflows.

The following Chapters 5 - 7 refine this abstract architecture into a detailed design that contains sufficient detail to construct a prototype implementation of the platform.
Chapter 5

Infrastructure

This chapter presents the detailed design of the Epsilon Model Connectivity and Epsilon Object Language components that refine the Model Connectivity and Core Language architectural layers discussed in Sections 4.4 and 4.5 respectively.

5.1 The Epsilon Model Connectivity Layer (EMC)

In this section the design of the Epsilon Model Connectivity layer discussed in Section 4.4 is presented. A graphical overview of the design is displayed in Figure 5.1
Figure 5.1: Overview of the Epsilon Model Connectivity layer
To abstract away from diverse model representations and APIs provided by different modelling technologies, EMC defines the \texttt{IModel} interface. \texttt{IModel} provides a number of methods that enable querying and modifying the model elements it contains at a higher level of abstraction. To enable languages and tools that build atop EMC to manage multiple models simultaneously, the \texttt{ModelRepository} class acts as a container that offers façade services. The following sections discuss these two core concepts in detail.

### 5.1.1 The \texttt{IModel} interface

Each model specifies a name which must be unique in the context of the model repository in which it is contained. Also, it defines a number of aliases; that is non-unique alternate names; via which it can be accessed. The interface also defines the following services.

#### Loading and Persistence

The \texttt{load()} and \texttt{load(properties : Properties)} methods enable extenders to specify in a uniform way how a model is loaded into memory from the physical location in which it resides. Similarly, the \texttt{store()} and \texttt{store(location : String)} methods are used to define how the model can be persisted from memory to a permanent storage location.

#### Type-related Services

The majority of metamodelling architectures support inheritance between meta-classes and therefore two types of type-conformance relationships generally appear between model elements and types. The \texttt{type-of} relationship appears when a model element is an instance of the type and the \texttt{kind-of} relationship appears when the model element is an instance of the type or any of its sub-types. Under this definition, the \texttt{getAllOfType(type : String)} and the \texttt{getAllOfKind(type : String)} operations return all the elements in the model that have a type-of and a kind-of relationship with the type in question respectively.
Similarly, the `isTypeOf(element : Object, type : String)` and `isKindOf(element : Object, type : String)` return whether the element in question has a type-of or a kind-of relationship with the type respectively. The `getTypeName(element : Object)` method returns the fully-qualified name of the type an element conforms to.

The `hasType(type : String)` method returns true if the model supports a type with the specified name. To support technologies that enable users to define abstract (non-instantiable) types, the `isInstantiable(type : String)` method returns if instances of the type can be created.

**Ownership**

The `allContents()` method returns all the elements that the model contains and the `owns(element : Object)` method returns true if the element under question belongs to the model.

**Creation, Deletion and Modifications**

Model elements are created and deleted using the `createInstance(type : String)` and `deleteElement(element : Object)` methods respectively.

To retrieve and set the values of properties of its model elements, `IModel` uses its associated `propertyGetter` (`IPropertyGetter`) and `propertySetter` (`IPropertySetter`) respectively. Technology-specific implementations of those two interfaces are responsible for accessing and modifying the value of a property of a model element through their `invoke(element:Object,property:String)` and `invoke(value:Object)` respectively.

**5.1.2 The IMoodelTransactionSupport interface**

In its `transactionSupport` property, a model can optionally (if the target modelling technology supports transactions) specify an instance of an implementation of the `IMoodelTransactionSupport` interface. The interface provides transaction-related services for the specific modelling technology. The interface provides the `startTransaction()`, `commitTransaction()` and `rollback-
Transaction() methods that start a new transaction, commit and roll back the current transaction respectively.

5.1.3 The ModelRepository class

A model repository acts as a container for a set of models that need to be managed in the context of a task or a set of tasks. Apart from a reference to the models it contains, ModelRepository also provides the following façade functionality.

The getOwningModel(element: Object) method returns the model that owns a particular element. The transactionSupport property specifies an instance of the ModelRepositoryTransactionSupport class which is responsible for aggregate management of transactions by delegating calls to its startTransaction(), commitTransaction() and abortTransaction() methods, to the respective methods of instances of IModelTransactionSupport associated with models contained in the repository.

5.1.4 The ModelGroup class

A ModelGroup is a group of models that have a common alias. ModelGroups are calculated dynamically by the model repository based on common model aliases. That is, if two or more models share a common alias, the repository forms a new model group. Since ModelGroup implements the IModel interface, clients can use all the methods of IModel to perform aggregate operations on multiple models, such as collecting the contents of more than one models. An exception to that is the createInstance(type: String) method which cannot be defined for a group of models as it cannot be determined in which model of the group the newly created element should belong.

5.1.5 Assumptions about the underlying modelling technologies

The discussion provided above has demonstrated that EMC makes only minimal assumptions about the structure and the organization of the underlying modelling technologies. Thus, it intentionally refrains from defining classes
for concepts such as *model element*, *type* and *metamodel*. By contrast, it employs a lightweight approach that uses primitive strings for type names and objects of the target implementation platforms as model elements. There are two reasons for this decision.

The primary reason is that by minimizing the assumptions about the underlying technologies EMC becomes more resistant to future changes of the implementations of the current technologies and can also embrace new technologies without changes.

Another reason is that if a heavy-weight approach was used, extending the platform with support for a new modelling technology would involve providing wrapping objects for the native objects which represent model elements and types in the specific modelling technology. Experiments in the early phases of the design of EMC demonstrated that such a heavy-weight approach significantly increases the amount of memory required to represent the models in memory, degrades performance and provides little benefits in reward\(^1\).

### 5.2 The Epsilon Object Language (EOL)

As discussed in Section 4.5, the primary aim of EOL is to provide a reusable set of common model management facilities, atop which task-specific languages can be implemented. However, EOL can also be used as a general-purpose standalone model management language for automating tasks that do not fall into the patterns targeted by task-specific languages. This section presents the syntax and semantics of the language using a combination of abstract syntax diagrams, concrete syntax examples and informal discussion.

Section 5.2.1 presents the rationale of the choice to discuss the semantics of the language informally as opposed to providing a formal semantics. Then, Sections 5.2.2 – 5.2.8 provide a detailed discussion on the syntax and the semantics of the language using abstract syntax diagrams to present

\(^1\)Recent developments in the context of the ATL transformation language have also demonstrated significant performance gains delivered by using native model element representations. Relevant benchmarks can be found [http://wiki.eclipse.org/ATL_VM_Testing](http://wiki.eclipse.org/ATL_VM_Testing)
structural aspects of the language, such as the structure of an EOL module and the underlying type system, and concrete syntax examples to demonstrate lower-level constructs, such as the different types of statements and expressions the language supports. A complete grammar that reflects the concrete syntax of EOL is provided in Appendix B.

5.2.1 Approaches to Specifying the Semantics of Languages

As discussed in [128, 129] there is a number of alternative approaches to defining the semantics of a language including denotational, operational, translational and pragmatic approaches. Expressing denotational semantics is achieved by constructing mathematical objects which represent the meaning of the constructs of the language. Operational semantics are expressed by describing how a program is interpreted as sequences of computational steps in a state transition system. The translational approach involves providing a mapping into another language that already has formal semantics. Finally, the pragmatic approach involves providing a reference implementation of the language in another executable language.

There are a number of reasons why in this work the latter approach has been chosen to express the semantics of the languages of the platform. As the work was carried out in the context of two large-scale EU integrated projects (ModelWare [40] and ModelPlex [39]), a reference implementation of the proposed languages was deemed necessary to enable industrial partners use the proposed languages to address practical problems. Also, by providing a reference implementation it was reasonably anticipated that substantial feedback from the community would be provided for the evolution of the languages of the platform.

Formal semantics would undoubtedly be useful to enable automated program analysis and verification and this is a particularly interesting subject for further research. However within the time frame of the thesis, both specifying formal semantics and providing a reference implementation was not possible. The decision to adopt a pragmtical approach by providing a reference implementation instead of defining a formal semantics is further
discussed and evaluated retrospectively in the evaluation part of this thesis in Chapter 9.

5.2.2 Module Organization

In this section the syntax of EOL is presented in a top-down manner. As displayed in Figure 5.2, EOL programs are organized in modules. Each module defines a body and a number of operations. The body is a block of statements that are evaluated when the module is executed. Each operation defines the kind of objects on which it is applicable (context), a name, a set of parameters and optionally a return type. Each module can also import other modules, using import statements, and access the operations they define.
Figure 5.2: EOL Module Structure
5.2.3 User-Defined Operations

In typical object-oriented languages such as Java and C++, operations are defined inside classes and can be invoked on instances of those classes. EOL on the other hand is not object-oriented in the sense that it does not define classes itself, but nevertheless needs to manage objects of types defined externally to it (e.g. in metamodels). By defining the context-type of an operation explicitly, the operation can be called on instances of the type as if it was natively defined by the type. Alternatively, context-less operations could be defined; however the adopted technique significantly improves readability of the concrete syntax.

For example, consider the code excerpts displayed in Listings 5.1 and 5.2. In Listing 5.1 the operations `add1` and `add2` are defined in the context of the built-in `Integer` type. Therefore, in line 1 they can be invoked using the `1.add1().add2().println()` expression. On the other hand, in Listing 5.2 where no context is defined, they have to be invoked in a nested manner which follows an in-to-out direction instead of the left to right direction used by the former excerpt. As complex model queries often involve invoking multiple properties and operations, this technique is particularly beneficial to the overall readability of the code.

Listing 5.1: Exemplar context-defining EOL operations

```plaintext
1. add1().add2().println();

operation Integer add1() : Integer {
    return self + 1;
}

operation Integer add2() : Integer {
    return self + 2;
}
```

Listing 5.2: Exemplar EOL context-less EOL operations

```plaintext
add2(add1(1)).println();

operation add1(base : Integer) : Integer {
```
EOL supports polymorphic operations using a runtime dispatch mechanism. Multiple operations with the same name and parameters can be defined, each defining a distinct context type. For example, in Listing 5.3, the statement in line 1 invokes the test operation defined in line 4, while the statement in line 2 invokes the test operation defined in line 8.

Listing 5.3: Demonstration of polymorphism in EOL

```plaintext
1 '1'.test();
2 1.test();
3
4 operation String test() {
5   (self + ' is a string').println();
6 }
7
8 operation Integer test() {
9   (self + ' is an integer').println();
10 }
```

Annotations

EOL supports two types of annotations: simple and executable. A simple annotation specifies a name and a set of String values while an executable annotation specifies a name and an expression. The concrete syntaxes of simple and executable annotations are displayed in Listings 5.4 and 5.5 respectively.

Listing 5.4: Concrete syntax of simple annotations

```plaintext
@name value(,value)*
```

Listing 5.5: Concrete syntax of executable annotations

```plaintext
$@name expression
```
In stand-alone EOL, annotations are supported only in the context of operations, however as discussed in the sequel, task-specific languages also make use of annotations in their constructs, each with task-specific semantics. EOL operations support three particular annotations: the \textit{pre} and \textit{post} executable annotations for specifying pre and post-conditions, and the \textit{cached} simple annotation, which are discussed below.

\textbf{Pre/post conditions in user-defined operations}

A number of \textit{pre} and \textit{post} executable annotations can be attached to EOL operations to specify the pre- and post-conditions of the operation. When an operation is invoked, before its body is evaluated, the expressions of the \textit{pre} annotations are evaluated. If all of them return \textit{true}, the body of the operation is processed, otherwise, an error is raised. Similarly, once the body of the operation has been executed, the expressions of the \textit{post} annotations of the operation are executed to ensure that the operation has had the desired effects. \textit{Pre} and \textit{post} annotations can access all the variables in the parent scope, as well as the parameters of the operation and the object on which the operation is invoked (through the \textit{self} variable). Moreover, in \textit{post} annotations, the returned value of the operation is accessible through the built-in \textit{result} variable. An example of using pre and post conditions in EOL appears in Listing 5.6.

\begin{verbatim}
Listing 5.6: Example of pre- and post-conditions in an EOL operation
1 l.add(2);
2 l.add(-1);
3 $pre i > 0
4 $post _result > self
5 operation Integer add(i : Integer) : Integer {
6    return self + i;
7 }

In line 4 the \textit{add} operation defines a pre-condition stating that the parameter \textit{i} must be a positive number. In line 5, the operation defines that result of the operation (\textit{result}) must be greater than the number on which
\end{verbatim}
it was invoked \(self\). Thus, when executed in the context of the statement in line 1, the operation succeeds, while when executed in the context of the statement in line 2, the pre-condition is not satisfied and an error is raised.

**Operation Result Caching**

EOL supports caching the results of parameter-less operations using the @cached simple annotation. In the following example, the Fibonacci number of a given Integer is calculated using the fibonacci recursive operation displayed in Listing 5.7. Since the fibonacci operation is declared as cached, it is only executed once for each distinct Integer and subsequent calls on the same target return the cached result. Therefore, when invoked in line 1, the body of the operation is called 16 times. By contrast, if no @cached annotation was specified, the body of the operation would be called recursively 1973 times. This feature is particularly useful for performing queries on large models and caching their results without needing to introduce explicit variables that store the cached results.

Listing 5.7: Calculating the Fibonacci number using a cached operation

```eol
  1. fibonacci().println();
  2.
  3 @cached
  4 operation Integer fibonacci() : Integer {
  5    if (self = 1 or self = 0) {
  6      return 1;
  7    }
  8    else {
  9      return (self-1).fibonacci() + (self-2).fibonacci();
 10    }
 11 }
```

**5.2.4 Types**

As is the case for most programming languages, EOL defines a built-in system of types, illustrated in Figure 5.3. The Any type, inspired by the OclAny type of OCL, is the basis of all types in EOL including Collection
types. The operations supported by instances of the Any type are outlined in Table 5.1.

\[\text{Parameters within square braces } [\ ] \text{ are optional}\]
Table 5.1: Operations of type Any

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isDefined() : Boolean</td>
<td>Returns true if the object is defined and false otherwise</td>
</tr>
<tr>
<td>isUndefined() : Boolean</td>
<td>Returns true if the object is undefined and false otherwise</td>
</tr>
<tr>
<td>isTypeOf(type : Type) : Boolean</td>
<td>Returns true if the object is of the given type and false otherwise</td>
</tr>
<tr>
<td>isKindOf(type : Type) : Boolean</td>
<td>Returns true if the object is of the given type or one of its subtypes and false otherwise</td>
</tr>
<tr>
<td>asString() : String</td>
<td>Returns a string representation of the object</td>
</tr>
<tr>
<td>asInteger() : Integer</td>
<td>Returns an Integer based on the string representation of the object. If the string representation is not of an acceptable format, an error is raised</td>
</tr>
<tr>
<td>asReal() : Real</td>
<td>Returns a Real based on the string representation of the object. If the string representation is not of an acceptable format, an error is raised</td>
</tr>
<tr>
<td>asBoolean() : Boolean</td>
<td>Returns a Boolean based on the string representation of the object. If the string representation is not of an acceptable format, an error is raised</td>
</tr>
<tr>
<td>asBag() : Bag</td>
<td>Returns a new Bag containing the object</td>
</tr>
<tr>
<td>asSequence() : Bag</td>
<td>Returns a new Sequence containing the object</td>
</tr>
<tr>
<td>asSet() : Set</td>
<td>Returns a new Set containing the object</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>asOrderedSet() : OrderedSet</td>
<td>Returns a new OrderedSet containing the object</td>
</tr>
<tr>
<td>print([prefix : String]) : Any</td>
<td>Prints a string representation of the object on which it is invoked prefixed with the optional <em>prefix</em> string and returns the object on which it was invoked. In this way, the <em>print</em> operation can be used for debugging purposes in a non-invasive manner</td>
</tr>
<tr>
<td>println([prefix : String]) : Any</td>
<td>Has the same effects with the <em>print</em> operation but also produces a new line in the output stream.</td>
</tr>
</tbody>
</table>
Primitives Types

EOL provides four primitive types: String, Integer, Real and Boolean. The String type represents finite sequences of characters and supports the following operations which can be invoked on its instances.

Table 5.2: Operations of type String

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>charAt(index : Integer) : String</td>
<td>Returns the character in the specified index</td>
</tr>
<tr>
<td>concat(str : String) : String</td>
<td>Returns a concatenated form of the string with the str parameter</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>toLowerCase() : String</td>
<td>Returns a new string where all the characters have been converted to lower case</td>
</tr>
<tr>
<td>firstToLowerCase() : String</td>
<td>Returns a new string the first character of which has been converted to lower case</td>
</tr>
<tr>
<td>toUpperCase() : String</td>
<td>Returns a new string where all the characters have been converted to upper case</td>
</tr>
<tr>
<td>firstToUpperCase() : String</td>
<td>Returns a new string, the first character of which has been converted to upper case</td>
</tr>
<tr>
<td>isSubstringOf(str : String) : Boolean</td>
<td>Returns true iff str is a sub-string of the string the operation is invoked on</td>
</tr>
<tr>
<td>matches(reg : String) : Boolean</td>
<td>Returns true if there are occurrences of the regular expression reg in the string</td>
</tr>
<tr>
<td>replace(source : String, target : String) : String</td>
<td>Returns a new string in which all instances of source have been replaced with instances of target</td>
</tr>
<tr>
<td>split(delimiter : String) : Sequence(String)</td>
<td>Splits the string based on the provided delimiter and returns a sequence containing the parts</td>
</tr>
<tr>
<td>startsWith(str : String) : Boolean</td>
<td>Returns true iff the string starts with str</td>
</tr>
<tr>
<td>substring(index : Integer) : String</td>
<td>Returns a sub-string of the string starting from the specified index and extending to the end of the original string</td>
</tr>
<tr>
<td>substring(index : Integer, size : Integer) : String</td>
<td>Returns a sub-string with the specified size, starting from the specified index</td>
</tr>
</tbody>
</table>

The Real type represents real numbers and provides the following operations.
Table 5.3: Operations of type Real

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ceiling() : Integer</td>
<td>Returns the nearest Integer that is larger than the real</td>
</tr>
<tr>
<td>floor() : Integer</td>
<td>Returns the nearest Integer that is greater than the real</td>
</tr>
<tr>
<td>abs() : Real</td>
<td>Returns the absolute value of the real</td>
</tr>
<tr>
<td>max(other : Real) : Real</td>
<td>Returns the maximum of the two reals</td>
</tr>
<tr>
<td>min(other : Real) : Real</td>
<td>Returns the minimum of the two reals</td>
</tr>
</tbody>
</table>

The Integer type represents natural numbers and negatives and extends the Real primitive type. Finally, the Boolean type represents true/false states and provides no additional operations to those provided by the base Any type.

**Collections and Maps**

EOL provides four types of collections and a Map type. The Bag type represents non-unique, unordered collections, the Sequence type represents non-unique, ordered collections, the Set type represents unique and unordered collections and the OrderedSet represents unique and ordered collections.

All collection types inherit from the abstract Collection type. Apart from simple operations, EOL also supports first-order logic operations on collections. The following operations apply to all types of collections:

Table 5.4: Operations of type Collection

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(item : Any)</td>
<td>Adds an item to the collection. If the collection is a set, addition of duplicate items has no effect</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>addAll(col : Collection)</code></td>
<td>Adds all the items of the <code>col</code> argument to the collection. If the collection is a set, it only adds items that do not already exist in the collection.</td>
</tr>
<tr>
<td><code>remove(item : Any)</code></td>
<td>Removes an <code>item</code> from the collection.</td>
</tr>
<tr>
<td><code>removeAll(col : Collection)</code></td>
<td>Removes all the items of <code>col</code> from the collection.</td>
</tr>
<tr>
<td><code>clear()</code></td>
<td>Empties the collection.</td>
</tr>
<tr>
<td><code>includes(item : Any) : Boolean</code></td>
<td>Returns <code>true</code> if the collection includes the <code>item</code>.</td>
</tr>
<tr>
<td><code>excludes(item : Any) : Boolean</code></td>
<td>Returns <code>true</code> if the collection excludes the <code>item</code>.</td>
</tr>
<tr>
<td><code>includesAll(col : Collection) : Boolean</code></td>
<td>Returns <code>true</code> if the collection includes all the items of collection <code>col</code>.</td>
</tr>
<tr>
<td><code>excludesAll(col : Collection) : Boolean</code></td>
<td>Returns <code>true</code> if the collection excludes all the items of collection <code>col</code>.</td>
</tr>
<tr>
<td><code>including(item : Any) : Collection</code></td>
<td>Returns a new collection that also contains the <code>item</code> – unlike the <code>add()</code> operation that adds the <code>item</code> to the collection itself.</td>
</tr>
<tr>
<td><code>excluding(item : Any) : Collection</code></td>
<td>Returns a new collection that excludes the <code>item</code> – unlike the <code>remove()</code> operation that removes the <code>item</code> from the collection itself.</td>
</tr>
<tr>
<td><code>includingAll(col : Collection) : Collection</code></td>
<td>Returns a new collection that is a union of the two collections. The type of the returned collection (i.e. Bag, Sequence, Set, OrderedSet) is same as the type of the collection on which the operation is invoked.</td>
</tr>
</tbody>
</table>
excludingAll(col: Collection): Collection

Returns a new collection that excludes all the elements of the col collection

flatten(): Collection

Recursively flattens all items that are of collection type and returns a new collection where no item is a collection itself

count(item: Any): Integer

Returns the number of times the item exists in the collection

size(): Integer

Returns the number of items the collection contains

isEmpty(): Boolean

Returns true if the collection does not contain any elements and false otherwise

random(): Any

Returns a random item from the collection

clone(): Collection

Returns a new collection of the same type containing the same items with the original collection

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>first(): Any</td>
<td>Returns the first item of the collection</td>
</tr>
<tr>
<td>last(): Any</td>
<td>Returns the last item of the collection</td>
</tr>
<tr>
<td>at(index: Integer): Any</td>
<td>Returns the item of the collection at the specified index</td>
</tr>
<tr>
<td>indexOf(item: Any): Integer</td>
<td>Returns the index of the item in the collection or -1 if it does not exist</td>
</tr>
</tbody>
</table>

The following operations apply to ordered collection types (i.e. Sequence and OrderedSet)

Table 5.5: Operations of types Sequence and OrderedSet

Also, EOL collection support the following first-order operations:
Table 5.6: First-order logic operations on Collections

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>select(iterator : Type</td>
<td>condition) : Collection</td>
</tr>
<tr>
<td>reject(iterator : Type</td>
<td>condition) : Collection</td>
</tr>
<tr>
<td>collect(iterator : Type</td>
<td>expression) : Collection</td>
</tr>
<tr>
<td>exists(iterator : Type</td>
<td>condition) : Boolean</td>
</tr>
<tr>
<td>forAll(iterator : Type</td>
<td>condition) : Boolean</td>
</tr>
<tr>
<td>iterate(iterator : IteratorType ; result : ResultType = initialValue</td>
<td>expression) : ResultType</td>
</tr>
<tr>
<td>sortBy(iterator : Type</td>
<td>expression) : Collection</td>
</tr>
</tbody>
</table>

The Map type represents an array of key-value pairs in which the keys are unique. The type provides the following operations.

Table 5.7: Operations of type Map
<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>put(key : Any, value : Any)</td>
<td>Adds the key-value pair to the map. If the map already contains the same key, the value is overwritten</td>
</tr>
<tr>
<td>get(key : Any) : Any</td>
<td>Returns the value for the specified keys</td>
</tr>
<tr>
<td>containsKey(key : Any) : Boolean</td>
<td>Returns true if the map contains the specified key</td>
</tr>
<tr>
<td>keys() : Set</td>
<td>Returns the keys of the map</td>
</tr>
<tr>
<td>values() : Bag</td>
<td>Returns the values of the map</td>
</tr>
<tr>
<td>clear()</td>
<td>Clears the map</td>
</tr>
</tbody>
</table>

### Native Types

As discussed earlier, while the purpose of EOL is to provide significant expressive power to enable users to manage models at a high level of abstraction, it is not intended to be a general-purpose programming language. Therefore, there may be cases where users need to implement some functionality that is either not efficiently supported by the EOL runtime (e.g. complex mathematical computations) or that EOL does not support at all (e.g. developing user interfaces, accessing databases). To overcome this problem, EOL enables users to create objects of the underlying programming environment by using native types. A native type specifies an implementation property that indicates the unique identifier for an underlying platform type. For instance, in a Java implementation of EOL the user can instantiate and use a Java class via its class identifier. Thus, in Listing 5.8 the EOL excerpt creates a Java window (Swing JFrame) and uses its methods to change its title and dimensions and make it visible.

Listing 5.8: Demonstration of NativeType in EOL

```eol
1 var frame := new Native('javax.swing.JFrame');
2 frame.title := 'Opened with EOL';
3 frame.setBounds(100,100,300,200);
4 frame.visible := true;
```
Model Element Types

A model element type represents a meta-level classifier. As discussed in Section 5.1, Epsilon intentionally refrains from defining more details about the meaning of a model element type to be able to support diverse modelling technologies where a type has different semantics. For instance a MOF class, an XSD complex type and a Java class can all be regarded as model element types according to the implementation of the underlying modelling framework.

In case of multiple models, as well as the name of the type, the name of the model is also required to resolve a particular type since different models may contain elements of homonymous but different model element types. In case a model defines more than one type with the same name (e.g. in different packages), a fully qualified type name must be provided.

In terms of concrete syntax, inspired by ATL, the ! character is used to separate the name of the type from the name of the model it is defined in. For instance Ma!A represents the type A of model Ma. Also, to support modelling technologies that provide hierarchical grouping of types (e.g. using packages) the :: notation is used to separate between packages and classes. A model element type supports the following operations:

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>allOfType() : Set</td>
<td>Returns all the elements in the model that are instances of the type</td>
</tr>
<tr>
<td>allOfKind() : Set</td>
<td>Returns all the elements in the model that are instances either of the type itself or of one of its subtypes</td>
</tr>
<tr>
<td>allInstances() : Set</td>
<td>Alias for allOfKind() (for compatibility with OCL)</td>
</tr>
<tr>
<td>all() : Set</td>
<td>Alias for allOfKind() (for syntax-compactness purposes)</td>
</tr>
</tbody>
</table>

Table 5.8: Operations of Model Element Types
isInstantiable() : Boolean
Returns true if the type is instantiable (i.e. non-abstract)

createInstance() : Any
Creates an instance of the type in the model

As an example of the concrete syntax, Listing 5.9 retrieves all the instances of the Class type (including instances of its subtypes) defined in the Core package of the UML 1.4 metamodel that are contained in the model named UML14.

Listing 5.9: Demonstration of the concrete syntax for accessing model element types

\[
\text{UML14!Core::Foundation::Class.allInstances();}
\]

5.2.5 Expressions

Feature Navigation

Since EOL needs to manage models defined using object oriented modelling technologies, it provides expressions to navigate properties and invoke simple and declarative operations on objects (as presented in Figure 5.4).

In terms of concrete syntax, ‘.’ is used as a uniform operator to access a property of an object and to invoke an operation on it. The ‘→’ operator, which is used in OCL to invoke first-order logic operations on sets, has been also preserved for syntax compatibility reasons. In EOL, every operation can be invoked both using the ‘.’ or the ‘→’ operators, with a slightly different semantics to enable overriding the built-in operations. If the ‘.’ operator is used, precedence is given to the user-defined operations, otherwise precedence is given to the built-in operations. For instance, the Any type defines a println() method that prints the string representation of an object to the standard output stream. In Listing 5.10, the user has defined another parameterless println() operation in the context of Any. Therefore the call to println() in Line 1 will be dispatched to the user-defined println()
operation defined in line 3. In its body the operation uses the ‘→’ operator to invoke the built-in println() operation (line 4).

Listing 5.10: Invoking operations using EOL

```
1  'Something'.println();
2
3  operation Any println() : Any { 
4      ('Printing : ' + self)->println(); 
5  }
```
Arithmetical and Comparison Operators

EOL provides common operators for performing arithmetical computations and comparisons illustrated in Tables 5.9 and 5.10 respectively.

Table 5.9: Arithmetical operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Adds reals/integers and concatenates strings</td>
</tr>
<tr>
<td>−</td>
<td>Subtracts reals/integers</td>
</tr>
<tr>
<td>− (unary)</td>
<td>Returns the negative of a real/integer</td>
</tr>
<tr>
<td>*</td>
<td>Multiplies reals/integers</td>
</tr>
<tr>
<td>/</td>
<td>Divides reals/integers</td>
</tr>
</tbody>
</table>

Table 5.10: Comparison operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Returns true if the left hand side equals the right hand side. In the case of primitive types (String, Boolean, Integer, Real) the operator compares the values; in the case of objects it returns true if the two expressions evaluate to the same object</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Is the logical negation of the (=) operator</td>
</tr>
<tr>
<td>&gt;</td>
<td>For reals/integers returns true if the left hand side is greater than the right hand side number</td>
</tr>
<tr>
<td>&lt;</td>
<td>For reals/integers returns true if the left hand side is less than the right hand side number</td>
</tr>
</tbody>
</table>
For reals/integers returns true if the left hand side is greater or equal to the right hand side number

For reals/integers returns true if the left hand side is less or equal to then right hand side number

### Logical Operators

EOL provides common operators for performing logical computations illustrated in Table 5.11. Logical operations apply only to instances of the Boolean primitive type.

#### Table 5.11: Logical Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>Returns the logical conjunction of the two expressions</td>
</tr>
<tr>
<td>or</td>
<td>Returns the logical disjunction of the two expressions</td>
</tr>
<tr>
<td>not</td>
<td>Returns the logical negation of the expression</td>
</tr>
<tr>
<td>implies</td>
<td>Returns the logical implication of the two expressions. Implication is calculated according to the truth table 5.12</td>
</tr>
<tr>
<td>xor</td>
<td>returns true if only one of the involved expressions evaluates to true and false otherwise</td>
</tr>
</tbody>
</table>

#### Table 5.12: Implies Truth Table

<table>
<thead>
<tr>
<th>Left</th>
<th>Right</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

111
5.2.6 Statements

Variable Declaration Statement

A variable declaration statement declares the name and (optionally) the type and initial value of a variable in an EOL program. If no type is explicitly declared, the variable is assumed to be of type Any. For variables of primitive type, declaration automatically creates an instance of the type with the default values presented in Table 5.13. For non-primitive types the user has to explicitly assign the value of the variable either by using the `new` keyword or by providing an initial value expression. If neither is done the value of the variable is undefined. Variables in EOL are strongly-typed. Therefore a variable can only be assigned values that conform to its type (or a sub-type of it).

Table 5.13: Default values of primitive types

<table>
<thead>
<tr>
<th>Type</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>0</td>
</tr>
<tr>
<td>Boolean</td>
<td>false</td>
</tr>
<tr>
<td>String</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>Real</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Scope  The scope of variables in EOL is generally limited to the block of statements where they are defined, including any nested blocks. Nevertheless, as discussed in the sequel, there are cases in task-specific languages that build atop EOL where the scope of variables is expanded to other non-nested blocks as well. EOL also allows variable shadowing; that is to define a variable with the same name in a nested block that overrides a variable...
In Listing 5.11, an example of declaring and using variables is provided. Line 1 defines a variable named \( i \) of type \( \text{Integer} \) and assigns it an initial value of 5. Line 2 defines a variable named \( c \) of type \( \text{Class} \) (from model Uml) and creates a new instance of the type in the model (by using the \textit{new} keyword). The commented out assignment statement of line 3 would raise a runtime error since it would attempt to assign a \textit{String} value to an \textit{Integer} variable. The condition of line 4 returns true since the \( c \) variable has been initialized before. Line 5 defines a new variable also named \( i \) that is of type \textit{String} and which overrides the \textit{Integer} variable declared in line 1. Therefore the assignment statement of line 6 is legitimate as it assigns a string value to a variable of type \textit{String}. Finally, as the program has exited the scope of the \textit{if} statement, the assignment statement of line 7 is also legitimate as it refers to the \( i \) variable defined in line 1.

**Listing 5.11: Example illustrating declaration and use of variables**

```
1 var i : Integer := 5;
2 var c : new Uml!Class;
3 --i := 'somevalue';
4 if (c.isDefined()) {
5     var i : String;
6     i := 'somevalue';
7 }
8 i := 3;
```

**Assignment Statement**

The assignment statement is used to update the values of variables and properties of native objects and model elements.

**Variable Assignment** When the left hand side of an assignment statement is a variable, the value of the variable is updated to the object to which the right hand side evaluates to. If the type of the right hand side is not compatible (kind-of relationship) with the type of the variable, the assignment is illegal and a runtime error is raised. Assignment to objects of
primitive types is performed by value while assignment to instances of non-primitive values is performed by reference. For example, in Listing 5.12, in line 1 the value of the a variable is set to a new Class in the Uml model. In line 2, a new untyped variable b is declared and its value is assigned to a. In line 3 the name of the class is updated to Customer and thus, line 4 prints Customer to the standard output stream. On the other hand, in Listing 5.13, in line 1 the a String variable is declared. In line 2 an untyped variable b is declared. In line 3, the value of a is changed to Customer (which is an instance of the primitive String type). This has no effect on b and thus line 4 prints an empty string to the standard output stream.

Listing 5.12: Assigning the value of a variable by reference

```java
var a : new Uml!Class;
var b := a;
a.name := 'Customer';
b.name.println();
```

Listing 5.13: Assigning the value of a variable by value

```java
var a : String;
var b := a;
a := 'Customer';
b.println();
```

**Native Object Property Assignment** When the left hand side of the assignment is a property of a native object, deciding on the legality and providing the semantics of the assignment is delegated to the execution engine. For example, in a Java-based execution engine, given that x is a native object, the statement x.y := a may be interpreted as x.setY(a) or if x is an instance of a map x.put("x",a). By contrast, in a C# implementation, it can be interpreted as x.y = a since the language natively supports properties in classes.

**Model Element Property Assignment** When the left hand side of the assignment is a property of a model element, the model that owns the particular model element (accessible using the ModelRepository.getOwningModel()
operation) is responsible for implementing the semantics of the assignment using its associated \textit{propertyGetter} as discussed in Section 5.1.1. For example, if \textit{x} is a model element, the statement \textit{x.y := a} may be interpreted using the Java code of Listing 5.14 if \textit{x} belongs to an EMF-based model or using the Java code of Listing 5.15 if it belongs to an MDR-based model.

Listing 5.14: Java code that assigns the value of a property of a model element that belongs to an EMF-based model

```
1 EStructuralFeature feature = x.eClass().getEStructuralFeature("y");
2 x.eSet(feature, a);
```

Listing 5.15: Java code that assigns the value of a property of a model element that belongs to an MDR-based model

```
1 StructuralFeature feature = findStructuralFeature(x.refClass(), "y");
2 x.refSetValue(feature, a);
```

**Special Assignment Statement**

In task-specific languages, an assignment operator with task-specific semantics is often required. Therefore, EOL provides an additional assignment operator. In standalone EOL, the operator has the same semantics with the primary assignment operator discussed above, however task-specific languages can redefine its semantics to implement custom assignment behaviour. For example, consider the simple model-to-model transformation of Listing 5.16 where a simple object oriented model is transformed to a simple database model using an ETL (see Section 6.2) transformation. The Class2Table rule transforms a Class of the OO model into a Table in the DB model and sets the name of the table to be the same as the name of the class. Rule Attribute2Column transforms an Attribute from the OO model into a column in the DB model. Except for setting its name (line 12), it also needs to define that the column belongs to the table which corresponds to the class that defines the source attribute. The commented-out assignment statement of line 13 cannot be used for this purpose since it would illegally attempt to assign the owningTable feature of the column to a model element of an inappropriate type (OO!Class). However, the special assignment
operator in the task-specific language implements the semantics discussed in Section 6.2.5 and thus in line 14 it assigns to the owningTable feature not the class that owns the attribute but its corresponding table (calculated using the Class2Table rule) in the DB model.

Listing 5.16: A simple model-to-model transformation demonstrating the special assignment statement

```plaintext
1 rule Class2Table
2   transform c : OO!Class to t : DB!Table {
3     t.name := c.name;
4   }
5
6 rule Attribute2Column
7   transform a : OO!Attribute to c : DB!Column {
8     c.name := a.name;
9     c.owningTable := c.owningClass;
10    c.owningTable ::= c.owningClass;
11   }
```

If Statement

As in most programming languages, an if statement consists of a condition, a block of statements that is executed if the condition is satisfied and (optionally) a block of statements that is executed otherwise. As an example, in Listing 5.17 if variable a holds a value that is greater than 0 the statement of line 3 is executed, otherwise the statement of line 5 is executed.

Listing 5.17: Example illustrating an if statement

```plaintext
1 if (a > 0) {
2    'A is greater than 0'.println();
3 } else {
4    'A is less equal than 0'.println();
5 }  
```

116
While Statement

A while statement consists of a condition and a block of statements which are executed as long as the condition is satisfied. For example, in Listing 5.18 the body of the while statement is executed 5 times printing the numbers 0 to 4 to the output console.

Listing 5.18: Example of a while statement

```plaintext
var i : Integer := 0;
while (i < 5) {
  i.println();
  i := i+1;
}
```

For Statement

In EOL, for statements are used to iterate the contents of collections. A for statement defines a typed iterator and an iterated collection as well as a block of statements that is executed for every item in the collection that has a kind-of relationship with the type defined by the iterator. As with the majority of programming languages, modifying a collection while iterating it raises a runtime error. To avoid this situation, users can use the clone() built-in operation of the Collection type discussed in 5.2.4.

Inside the body of `for` statements two built-in read-only variables are visible: the hasMore boolean variable is used to determine if there are more items if the collection for which the loop will be executed and the loopCount integer variable holds the number of times the innermost loop has been executed so far (including the current iteration). For example, in Listing 5.19 the `col` heterogeneous Sequence is defined that contains two strings (a and b), two integers (1,2) and one real (2.5). The for loop of line 2 only iterates through the items of the collection that are of kind Real and therefore prints 1,2,2.5 to the standard output stream.

Listing 5.19: Example of a for statement

```plaintext
var col : Sequence := Sequence{'a', 1, 2, 2.5, 'b'};
for (r : Real in col) {
  ...
}
```
for (i in Sequence{1..3}) {
    if (i = 1) {continue;}
    for (j in Sequence{1..4}) {
        if (j = 2) {break;}
        if (j = 3) {breakAll;}
        (i + ', ' + j).println();
    }
}

Transaction Statement

As discussed in Section 5.1.2 the underlying EMC layer provides support for transactions in models. To utilize this feature EOL provides the transaction statement. A transaction statement (optionally) defines the models that participate in the transaction. If no models are defined, it is assumed that all the models that are accessible from the enclosing program participate. When the statement is executed, a transaction is started on each participating model. If no errors are raised during the execution of the contained statements, any changes made to model elements are committed. On the other hand, if an error is raised the transaction is rolled back and any changes made to the models in the context of the transaction are undone. The user can also use the abort statement to explicitly exit a transaction.
and roll-back any changes done in its context. In Listing 5.21, an example of using this feature in a simulation problem is illustrated.

In this problem, a system consists of a number of processors. A processor manages some tasks and can fail at any time. The EOL program in Listing 5.21 performs 100 simulation steps, in every one of which 10 random processors from the model (lines 7-11) are marked as failed by setting their failed property to true (line 14). Then, the tasks that the failed processors manage are moved to other processors (line 15). Finally the availability of the system in this state is evaluated.

After a simulation step, the state of the model has been drastically changed since processors have failed and tasks have been relocated. To be able to restore the model to its original state after every simulation step, each step is executed in the context of a transaction which is explicitly aborted (line 20) after evaluating the availability of the system. Therefore after each simulation step the model is restored to its original state for the next step to be executed.

Listing 5.21: Example of a for statement

```eol
var system : System.allInstances.first();

for (i in Sequence {1..100}) {
  transaction {
    var failedProcessors : Set;

    while (failedProcessors.size() < 10) {
      failedProcessors.add(system.processors.random());
    }

    for (processor in failedProcessors) {
      processor.failed := true;
      processor.moveTasksElsewhere();
    }

    system.evaluateAvailability();
  }
}
```
5.2.7 Extended Properties

Quite often, during a model management operation it is necessary to associate model elements with information that is not supported by the metamodel they conform to. For instance, the EOL program in listing 5.22 calculates the depth of each Tree element in a model that conforms to the Tree metamodel displayed in Figure 5.5.

Listing 5.22: Calculating and printing the depth of each Tree

```java
var depths := new Map;

for (n in Tree.allInstances.select(t|not t.parent.isDefined())) {
    n.setDepth(0);
}

for (n in Tree.allInstances) {
    (n.name + ' ' + depths.get(n)).println();
}

operation Tree setDepth(depth : Integer) {
    depths.put(self, depth);
    for (c in self.children) {
        c.setDepth(depth + 1);
    }
}
```

As the Tree metamodel doesn’t support a depth property in the Tree metaclass, each Tree has to be associated with its calculated depth (line 12) using the depths map defined in line 1. Another approach would be to extend the Tree metamodel to support the desired depth property; however, applying this technique every time an additional property is needed for some model management operation would quickly pollute the metamodel with properties of secondary importance.
Figure 5.5: The Tree Metamodel

To simplify the code required in such cases, EOL provides the concept of extended properties. In terms of concrete syntax, an extended property is a normal property, the name of which starts with the tilde character (~). With regards to its execution semantics, the first time the value of an extended property of an object is assigned, the property is created and associated with the object. Then, the property can be accessed as a normal property. Listing 5.23 demonstrates using a depth extended property to eliminate the need for using the depths map in Listing 5.22.

Listing 5.23: A simplified version of Listing 5.22 using extended properties

```plaintext
for (n in Tree.allInstances.select(t|not t.parent.isDefined())) {
    n.setDepth(0);
}

for (n in Tree.allInstances) {
    (n.name + ' ' + n."depth").println();
}

operation Tree setDepth(depth : Integer) {
    self."depth" := depth;
    for (c in self.children) {
        c.setDepth(depth + 1);
    }
}
```
5.2.8 Context-Independent User Input

A common assumption in model management languages is that model management tasks are only executed in a batch-manner without human intervention. However, as demonstrated in the sequel, it is often useful for the user to provide feedback that can precisely drive the execution of a model management operation.

Model management operations can be executed in a number of runtime environments in each of which a different user-input method is more appropriate. For instance when executed in the context of an IDE (such as Eclipse) visual dialogs are preferable, while when executed in the context of a server or from within an ANT workflow, a command-line user input interface is deemed more suitable. To abstract away from the different runtime environments and enable the user to specify user interaction statements uniformly and regardless of the runtime context, EOL provides the `IUserInput` interface that can be realized in different ways according to the execution environment and attached to the runtime context via the `IEolContext.setUserInput(IUserInput userInput)` method. The `IUserInput` specifies the methods presented in Table 5.14.

Table 5.14: Operations of IUserInput

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inform(message : String)</td>
<td>Displays the specified message to the user</td>
</tr>
<tr>
<td>confirm(message : String, default : Boolean) : Boolean</td>
<td>Prompts the user to confirm if the condition described by the message holds</td>
</tr>
<tr>
<td>prompt(message : String, default : String) : String</td>
<td>Prompts the user for a string in response to the message</td>
</tr>
<tr>
<td>promptInteger(message : String, default : Integer) : Integer</td>
<td>Prompts the user for an Integer</td>
</tr>
<tr>
<td>promptReal(message : String, default : Real) : Real</td>
<td>Prompts the user for a Real</td>
</tr>
</tbody>
</table>
choose(message : String, options : Sequence, default : Any) : Any
Prompts the user to select one of the options

chooseMany(message : String, options : Sequence, default : Sequence) : Sequence
Prompts the user to select one of the options

As displayed above, all the methods of the IUserInput interface accept a default parameter. The purpose of this parameter is dual. First, it enables the designer of the model management program to prompt the user with the most likely value as a default choice and secondly it enables a concrete implementation of the interface (UnattendedExecutionUserInput) which returns the default values without prompting the user at all and thus, can be used for unattended execution of interactive Epsilon programs. Figures 5.6 and 5.7 demonstrate the interfaces through which input is required by the user when the exemplar System.user.promptInteger('Please enter a number', 1); statement is executed using an Eclipse-based and a command-line-based IUserInput implementation respectively.

User-input facilities have been found to be particularly useful in all model management tasks. Such facilities are essential for performing operations on live models such as model validation and model refactoring but can also be useful in model comparison where marginal matching decisions can be delegated to the user and model transformation where the user can interactively

Figure 5.6: Example of an Eclipse-based IUserInput implementation
specify the elements that will be transformed into corresponding elements in the target model. Examples of interactive model management operations that make use of the input facilities provided by EOL are demonstrated in Sections 6.2.6 and 6.1.5.

5.2.9 Novel Features

As EOL is an amalgam of standard and novel features, the purpose of this section is to distinguish its unique features from standard features inspired or reused by existing model management languages.

Novel features of EOL include support for extended properties discussed in Section 5.2.7, support for context-independent user interaction discussed in Section 5.2.8, support for transactions in model management programs discussed in Section 5.2.6, the provision of a special assignment operator (:=) that can be overloaded by higher order languages with task-specific semantics discussed in Section 5.2.6, and support for cached operations discussed in Section 5.2.3. The remaining features of the language have been largely inspired from existing model management and programming languages; mainly from OCL, ATL, MOFScript and Java.
5.3 Chapter Summary

This chapter has provided a detailed discussion on the infrastructure atop which task-specific model management languages can be built. The infrastructure consists of the Epsilon Model Connectivity component which is responsible for providing a uniform interface for accessing models of diverse metamodels and modelling technologies, and the Epsilon Object Language which provides core language constructs for programmatic interaction with models. The next chapter provides a detailed discussion on the task-specific languages that have been designed atop this infrastructure.
Chapter 6

Task-Specific Languages

In this chapter, the following task-specific languages built atop the infrastructure discussed in Chapter 5 are presented:

- Epsilon Comparison Language (ECL)
- Epsilon Transformation Language (ETL)
- Epsilon Merging Language (EML)
- Epsilon Validation Language (EVL)
- Epsilon Wizard Language (EWL)

For each language, the abstract and concrete syntax are presented. To enhance readability, the concrete syntax of each language is presented in an abstract, pseudo-grammar form, and the complete formal grammars of the languages are available in Appendix B. Also provided is an informal (see Section 5.2.1) but detailed discussion, accompanied by concise examples for each feature of interest, of its execution semantics and the runtime structures that are essential to implement those semantics.

Descriptions of the abstract and concrete syntaxes of the task-specific languages are particularly brief since they inherit most of their syntax and features from EOL. As discussed earlier, this contributes to establishing
a platform of uniform languages where each provides a number of unique task-specific constructs but does not otherwise deviate from each other.

To reduce unnecessary repetition, the following sections do not repeat all the features inherited from EOL. However, the reader should bear in mind that by being supersets of EOL, all task-specific languages can exploit the features it provides. For example, by reusing EOL’s user-input facilities (discussed in 5.2.8), it is feasible to specify interactive model to model transformations in ETL. As well, Native types can be used to access or update information stored in an external system/tool (e.g. in a database or a remote server) during model validation with EVL or model comparison with ECL.

Following the presentation, in Sections 6.1 – 6.5 of the task-specific languages implemented in this thesis, Section 6.7 provides a brief overview of the process needed to construct a new language that addresses a task that is not supported by one of the existing languages.
6.1 The Epsilon Comparison Language (ECL)

As discussed in Section 2.4.3, model comparison is the task of identifying matching elements between models. In general, matching elements are elements that are involved in a relationship of interest. For example, before merging homogeneous models, it is essential to identify overlapping (common) elements so that they do not appear in duplicate in the merged model. Similarly, in heterogeneous model merging, it is a prerequisite to identify the elements on which the two models will be merged. Finally, in transformation testing, matching elements are pairs consisting of elements in the input model and their generated counterparts in the output model.

The aim of the Epsilon Comparison Language (ECL) is to enable users to specify comparison algorithms in a rule-based manner to identify pairs of matching elements between two models of potentially different metamodels and modelling technologies. In this section, the abstract and concrete syntax, as well as the execution semantics of the language, are discussed in detail.

6.1.1 Abstract Syntax

In ECL, comparison specifications are organized in modules (EcLModule). As illustrated in Figure 6.1, EcLModule extends EOLLibraryModule which means that it can contain user-defined operations and import other library modules and ECL modules. Apart from operations, an ECL module contains a set of match-rules (MatchRule) and a set of pre and post blocks.

MatchRules enable users to perform comparison of model elements at a high level of abstraction. Each match-rule declares a name, and two parameters (leftParameter and rightParameter) that specify the types of elements it can compare. It also optionally defines a number of rules it inherits (extends) and if it is abstract, lazy and/or greedy. The semantics of the latter are discussed shortly.
Figure 6.1: ECL Abstract Syntax
A match rule has three parts. The guard part is an EOL expression or statement block that further limits the applicability of the rule to an even narrower range of elements than that specified by the left and right parameters. The compare part is an EOL expression or statement block that is responsible for comparing a pair of elements and deciding if they match or not. Finally, the do part is an EOL expression or block that is executed if the compare part returns true to perform any additional actions required.

Pre and Post blocks are named blocks of EOL statements which as discussed in the sequel are executed before and after the match-rules have been executed respectively.

6.1.2 Concrete Syntax

The concrete syntax of a match-rule is displayed in Listing 6.1.

```
Listing 6.1: Concrete Syntax of a MatchRule

(@lazy)?
(@greedy)?
(@abstract)?
ru<name>
match <leftParameterName>:<leftParameterType>
with <rightParameterName>:<rightParameterType>
extends (<ruleName>,)*<ruleName>)? {
  (guard (:expression)|{{statementBlock}})?
  compare (:expression)|{{statementBlock}}
  (do {statementBlock})?
}
```

Pre and post blocks have a simple syntax that, as presented in Listing 6.2, consists of the identifier (pre or post), an optional name and the set of statements to be executed enclosed in curly braces.
Listing 6.2: Concrete Syntax of Pre and Post blocks

1 (pre|post) <name> {  
2    statement+  
3 }

6.1.3 Execution Semantics

Rule and Block Overriding

An ECL module can import a number of other ECL modules. In such a case, the importing ECL module inherits all the rules and pre/post blocks specified in the modules it imports (recursively). If the module specifies a rule or a pre/post block with the same name, the local rule/block overrides the imported one respectively.

Comparison Outcome

As illustrated in Figure 6.2, the result of comparing two models with ECL is a trace (MatchTrace) that consists of a number of matches (Match). Each match holds a reference to the objects from the two models that have been compared (left and right), a boolean value that indicates if they have been found to be matching or not, a reference to the rule that has made the decision, and a Map (info) that is used to hold any additional information required by the user. During the matching process, a second, temporary, match trace is also used to detect and resolve cyclic invocation of match-rules as discussed in the sequel.

Rule Execution Scheduling

Non-abstract, non-lazy match-rules are evaluated automatically by the execution engine in a top-down fashion - with respect to their order of appearance - in two passes. In the first pass, each rule is evaluated for all the pairs of instances in the two models that have a type-of relationship with the types specified by the leftParameter and rightParameter of the rule. In the second pass, each rule that is marked as greedy is executed for all pairs
that have not been compared in the first pass, and which have a kind-of
relationship with the types specified by the rule. In both passes, to evaluate
the compare part of the rule, the guard must be satisfied.

Before the compare part of a rule is executed, the compare parts of all
of the rules it extends (super-rules) must be executed (recursively). Before
executing the compare part of a super-rule, the engine verifies that the super-
rule is actually applicable to the elements under comparison by checking for
type conformance and evaluating the guard part of the super-rule.

If the compare part of a rule evaluates to true, the optional do part is
executed. In the do part the user can specify any actions that need to be performed for the identified matching elements, such as to populate the info map of the established match with additional information. Finally, a new match is added to the match trace that has its matching property set to the logical conjunction of the results of the evaluation of the compare parts of the rule and its super-rules.

The matches() built-in operation

To refrain from performing duplicate comparisons and to de-couple match-rules from each other, ECL provides the built-in matches(opposite : Any) operation for model elements and collections. When the match operation is invoked on a pair of objects, it queries the main and temporary match-traces to discover if the two elements have already been matched and if so it returns the cached result of the comparison. Otherwise, it attempts to find an appropriate match rule to compare the two elements and if such a rule is found, it returns the result of the comparison, otherwise it returns false. Unlike the top-level execution scheme, the matches() operation invokes both lazy and non-lazy rules.

In addition to objects, the matches operations can also be invoked to match pairs of collections of the same type (e.g. a Sequence against a Sequence). When invoked on ordered collections (i.e. Sequence and OrderedSet), it examines if the collections have the same size and each item of the source collection matches with the item of the same index in the target collection. Finally, when invoked on unordered collections (i.e. Bag and Set), it examines if for each item in the source collection, there is a matching item in the target collection irrespective of its index. Users can also override the built-in matches operation using user-defined operations with the same name, as discussed in Section 5.2.5 that loosen or strengthen the built-in semantics.
Cyclic invocation of matches()

Providing the built-in matches operation significantly simplifies comparison specifications. It also enhances decoupling between match-rules from each other as when a rule needs to compare two elements that are outside its scope, it does not need to know/specify which other rule can compare those elements explicitly.

On the other hand, it is possible - and quite common indeed - for two rules to implicitly invoke each other. For example consider the match rule of Listing 6.3 that attempts to match nodes of the simple Tree metamodel displayed in Figure 6.3.

```
rule Tree2Tree
match l : T1!Tree
with r : T2!Tree {
  compare : l.label = r.label and
           l.parent.matches(r.parent) and
           l.children.matches(r.children)
}
```

The rule specifies that for two Tree nodes (l and r) to match, they should have the same label, belong to matching parents and have matching children. In the absence of a dedicated mechanism for cycle detection and resolution, the rule would end up in an infinite loop. To address this problem, ECL provides a temporary match-trace which is used to detect and resolve cyclic invocations of the match() built-in operation.
As discussed above, a match is added to the primary match-trace as soon as the compare part of the rule has been executed to completion. By contrast, a temporary match (with its matching property set to true) is added to the temporary trace before the compare part is executed. In this way, any subsequent attempts to match the two elements from invoked rules will not re-invoke the rule. Finally, when a top-level rule returns, the temporary match trace is reset.

6.1.4 Fuzzy and Dictionary-based String Matching

In the example of Listing 6.3, the rule specifies that to match, two trees must - among other criteria - have the same label. However, there are cases when a less-strict approach to matching string properties of model elements is desired. For instance, when comparing two UML models originating from different organizations, it is common to encounter ontologically equivalent classes which however have different names (e.g. Client and Customer). In this case, to achieve a more sound matching, the use of a dictionary or a lexical database (e.g. WordNet [130]) is necessary. Alternatively, fuzzy string matching algorithms such as those presented in [131] can be used.

As several such tools and algorithms have been implemented in various programming languages, it is a sensible approach to reuse them instead of re-implementing them. For example, in Listing 6.4 a wrapper for the Simmetrics [132] fuzzy string comparison tool is used to compare the labels of the trees using the Levenshtein [133] algorithm. To achieve this, line 11 invokes the fuzzyMatch() operation defined in lines 16-18 which uses the simmetrics native tool (instantiated in lines 2-4) to match the two labels using their Levenshtein distance with a threshold of 0.5.

Listing 6.4: The FuzzyTree2Tree rule

```plaintext
pre {
    var simmetrics :=
        new Native('org.epsilon.ecl.tools.
            textcomparison.simmetrics.SimMetricsTool');
}
```
rule FuzzyTree2Tree
match l : T1!Tree
with r : T2!Tree {
  compare : l.label.fuzzyMatch(r.label) and
  l.parent.matches(r.parent) and
  l.children.matches(r.children)
}

operation String fuzzyMatch(other : String) : Boolean {
  return simmetrics.similarity(self,other,'Levenshtein') > 0.5;
}

6.1.5 Interactive Matching

Using the user interaction features discussed in Section 5.2.8 the comparison can become interactive by replacing the fuzzyMatch operation of listing 6.4 with the one specified in Listing 6.5. The fuzzyMatch operation of Listing 6.5 performs the fuzzy string comparison and – as the previous version – if the result is greater than 0.5 it returns true. However, in this updated version if the result is lower than 0.5 but greater than 0.3, it prompts the user to confirm if the two strings match, and if it is lower than 0.3 it returns false.

Listing 6.5: An interactive version of the fuzzyMatch operation of Listing 6.4

operation String fuzzyMatch(other : String) : Boolean {
  var similarity : Real;
  similarity := simmetrics.similarity(self,other,'Levenshtein');
  if (similarity > 0.5) {
    return true;
  }
  else if (similarity > 0.3) {
    return UserInput.confirm(self + ' matches ' + other + '?');
  }
  else {
    return false;
  }
}
6.1.6 Exploiting the Comparison Outcome

Users can query and modify the match trace calculated during the comparison process in the post sections of the module or export it into another application or Epsilon program. For example, in a post section, the trace can be printed to the default output stream or serialized into a model of an arbitrary metamodel. In another use case, the trace may be exported to be used in the context of a validation module that will use the identified matches to evaluate inter-model constraints, or in a merging module that will use the matches to identify the elements on which the two models will be merged. The topic of interoperability - that includes importing and exporting objects - between modules expressed in different Epsilon languages is discussed in Chapter 7.
6.2 The Epsilon Transformation Language (ETL)

The aim of ETL [15] is to contribute model-to-model transformation capabilities to Epsilon. More specifically, ETL can be used to transform an arbitrary number of input models into an arbitrary number of output models of different modelling languages and technologies at a high level of abstraction.

6.2.1 Style

Summarizing the detailed discussion provided in Section 2.4.2, three styles are generally recognized in model transformation languages: declarative, imperative and hybrid, each one demonstrating particular advantages and shortcomings. Declarative transformation languages are generally limited to scenarios where the source and target metamodels are similar to each other in terms of structure and thus, the transformation is a matter of a simple mapping. However they fail to address cases where significant processing and complex mappings are involved. On the other hand, purely imperative transformation languages are capable of addressing a wider range of transformation scenarios. Nevertheless, they operate at a low level of abstraction which means that users need to manually address issues such as tracing and resolving target elements from their source counterparts and orchestrating the transformation execution. To address those shortcomings, hybrid languages (such as ATL [83] and QVT [82]) provide both a declarative rule-based execution scheme as well as imperative features for handling complex transformation scenarios.

Under this rationale, ETL has been designed as a hybrid language that implements a task-specific rule definition and execution scheme but also inherits the imperative features of EOL to handle complex transformations where this is deemed necessary.
6.2.2 Source and Target Models

The majority of model-to-model transformation languages assume that only two models participate in each transformation: the source model and the target model. Nevertheless, it is often essential to be able to access/update additional models during a transformation (such as trace or configuration models). Building on the facilities provided by EMC and EOL, ETL enables specification of transformations that can transform an arbitrary number of source models into an arbitrary number of target models.

Another common assumption is that the contents of the target models are insignificant and thus a transformation can safely overwrite its contents. As discussed in the sequel, ETL - like all Epsilon languages - enables the user to specify, for each involved model, whether its contents need to be preserved or not.

6.2.3 Abstract Syntax

As illustrated in Figure 6.4, ETL transformations are organized in modules (EtlModule). A module can contain a number of transformation rules (TransformationRule). Each rule has a unique name (in the context of the module) and also specifies one source and many target parameters. A transformation rule can also extend a number of other transformation rules and be declared as abstract, primary and/or lazy\(^1\). To limit its applicability to a subset of elements that conform to the type of the source parameter, a rule can optionally define a guard which is either an EOL expression or a block of EOL statements. Finally, each rule defines a block of EOL statements (body) where the logic for populating the property values of the target model elements is specified.

Besides transformation rules, an ETL module can also optionally contain a number of pre and post named blocks of EOL statements which, as discussed later, are executed before and after the transformation rules respectively.

\(^1\)The concept of lazy rules was first introduced in ATL
Figure 6.4: ETL Abstract Syntax
6.2.4 Concrete Syntax

The concrete syntax of a transformation rule is displayed in Listing 6.6.

The optional abstract, lazy and primary attributes of the rule are specified using respective annotations. The name of the rule follows the rule keyword and the source and target parameters are defined after the transform and to keywords. Also, the rule can define an optional comma-separated list of rules it extends after the extends keyword. Inside the curly braces ({}), the rule can optionally specify its guard either as an EOL expression following a colon (:) (for simple guards) or as a block of statements in curly braces (for more complex guards). Finally, the body of the rule is specified as a sequence of EOL statements.

Listing 6.6: Concrete Syntax of a TransformationRule

```
1 (@abstract)?
2 (@lazy)?
3 (@primary)?
4 rule <name>
5    transform <sourceParameterName>::<sourceParameterType>
6    to (<rightParameterName>::<rightParameterType>
7        , <rightParameterName>::<rightParameterType>)*
8    (extends (<ruleName>,)*<ruleName>)? { 
9
10   (guard (:expression)|({statement+}))? 
11
12   statement+
13 }
```

Pre and post blocks have a simple syntax that, as presented in Listing 6.7, consists of the identifier (pre or post), an optional name and the set of statements to be executed enclosed in curly braces.

Listing 6.7: Concrete Syntax of Pre and Post blocks

```
1 (pre|post) <name> { 
2   statement+
3 }
```
6.2.5 Execution Semantics

Rule and Block Overriding

Similarly to ECL, an ETL module can import a number of other ETL modules. In this case, the importing ETL module inherits all the rules and pre/post blocks specified in the modules it imports (recursively). If the module specifies a rule or a pre/post block with the same name, the local rule/block overrides the imported one respectively.

Rule Execution Scheduling

When an ETL module is executed, the \textit{pre} blocks of the module are executed first in the order in which they have been specified.

Following that, each \textit{non-abstract} and \textit{non-lazy} rule is executed for all the elements on which it is applicable. To be applicable on a particular element, the element must have a kind-of relationship with the type defined in the rule’s \textit{sourceParameter} and must also satisfy the \textit{guard} of the rule (and all the rules it extends). When a rule is executed on an applicable element, the target elements are initially created by instantiating the \textit{targetParameters} of the rules, and then their contents are populated using the EOL statements of the \textit{body} of the rule.

Finally, when all rules have been executed, the \textit{post} blocks of the module are executed in the order in which they have been declared.

Source Elements Resolution

Resolving target elements that have (or can be) transformed from source elements by other rules is a frequent task in the body of a transformation rule. To automate this task and reduce coupling between rules, ETL contributes the \textit{equivalents()} and \textit{equivalent()} built-in operations that automatically resolve source elements to their transformed counterparts in the target models.

When the \textit{equivalents()} operation is applied on a single source element (as opposed to a collection of them), it inspects the established transfor-
information trace (displayed in Figure 6.6) and invokes the applicable rules (if necessary) to calculate the counterparts of the element in the target model. When applied to a collection it returns a Bag containing Bags that in turn contain the counterparts of the source elements contained in the collection. The equivalents() operation can be also invoked with an arbitrary number of rule names as parameters to invoke and return only the equivalents created by specific rules. Unlike the main execution scheduling scheme discussed above, the equivalents() operation invokes both lazy and non-lazy rules.

With regard to the ordering of the results of the equivalents() operations, the returned elements appear in the respective order of the rules that have created them. An exception to this occurs when one of the rules is declared as primary, in which case its results precede the results of all other rules.

ETL also provides the convenience equivalent() operation which, when applied to a single element, returns only the first element of the respective result that would have been returned by the equivalents() operation discussed above. Also, when applied to a collection the equivalent() operation returns a flattened collection (as opposed to the result of equivalents() which is a Bag of Bags in this case). As with the equivalents() operation, the equivalent() operation can also be invoked with or without parameters.

The semantics of the equivalent() operation are further illustrated through a simple example. In this example, we need to transform a model that conforms to the Tree metamodel displayed in Figure 6.3 into a model that conforms to the Graph metamodel of Figure 6.5. More specifically, we need to transform each Tree element to a Node, and an Edge that connects it with the Node that is equivalent to the tree’s parent. This is achieved using the rule of Listing 6.8.

In lines 1-3, the Tree2Node rule specifies that it can transform elements of the Tree type in the Tree model into elements of the Node type in the Graph model. In line 4 it specifies that the name of the created Node should be the same as the name of the source Tree. If the parent of the source Tree is defined (line 7), the rule creates a new Edge (line 8) and sets its source property to the created Node (line 9) and its target property to the equivalent Node of the source Tree’s parent (line 10).
Listing 6.8: Exemplar ETL Rule demonstrating the `equivalent()` operation

```java
rule Tree2Node
    transform t : Tree!Tree
    to n : Graph!Node {
        n.label := t.label;
        if (t.parent.isDefined()) {
            var edge := new Graph!Edge;
            edge.source := n;
            edge.target := t.parent.equivalent();
        }
    }
```

Figure 6.5: A Simple Graph Metamodel
Figure 6.6: ETL Runtime
Overriding the semantics of the EOL SpecialAssignmentOperator

As discussed above, resolving the equivalent(s) or source model elements in the target model is a recurring task in model transformation. Furthermore, in most cases resolving the equivalent of a model element is immediately followed by assigning/adding the obtained target model elements to the value(s) of a property of another target model element. For example, in line 10 of Listing 6.8 the equivalent obtained is immediately assigned to the target property of the generated Edge. To make transformation specifications more readable, ETL overrides the semantics of the SpecialAssignmentStatement (::= in terms of concrete syntax), described in Section 5.2.6 to set its left-hand side, not to the element its right-hand side evaluates to, but to its equivalent as calculated using the equivalent() operation discussed above. Using this feature, line 10 of the Tree2Node rule can be rewritten as shown in Listing 6.9.

Listing 6.9: Rewritten Line 10 of the Tree2Node Rule Demonstrated in Listing 6.8

```plaintext
edge.target ::= t.parent;
```

6.2.6 Interactive Transformations

Using the user interaction facilities of EOL discussed in Section 5.2.8 an ETL transformation can become interactive by prompting the user for input during its execution. For example in Listing 6.10 we modify the Tree2Node rule originally presented in Listing 6.8 by adding a guard part that uses the user-input facilities of EOL (more specifically the UserInput.confirm(String,Boolean) operation) to enable the user select manually at runtime which of the Tree elements need to be transformed to respective Node elements in the target model and which not.

Listing 6.10: Exemplar Interactive ETL Transformation

```plaintext
rule Tree2Node
   transform t : Tree!Tree
   to n : Graph!Node { 
```
6.2.7 Summary

This section has provided a detailed discussion on the Epsilon Transformation Language (ETL). ETL is capable of transforming an arbitrary number of source models into an arbitrary number of target models. ETL adopts a hybrid style and features declarative rule specification using advanced concepts such as guards, abstract, lazy and primary rules, and automatic resolution of target elements from their source counterparts. Also, as ETL is based on EOL reuses its imperative features to enable users to specify particularly complex, and even interactive, transformations.
6.3 The Epsilon Validation Language (EVL)

The aim of EVL is to contribute model validation capabilities to Epsilon. More specifically, EVL can be used to specify and evaluate constraints on models of arbitrary metamodels and modelling technologies. This section provides a discussion on the motivation for implementing EVL, its abstract and concrete syntax as well as its execution semantics. It also provides two examples using the language to verify inter-model and intra-model consistency.

6.3.1 Motivation

Although many approaches have been proposed to enable automated model validation, the Object Constraint Language (OCL) \[57\] is the de facto standard for capturing constraints in modelling languages specified using object-oriented metamodeling technologies. While its powerful syntax enables users to specify meaningful and concise constraints, its purely declarative and side-effect free nature introduces a number of limitations in the context of a contemporary model management environment. In this section, the shortcomings of OCL that have motivated the design of EVL are discussed in detail.

In OCL, structural constraints are captured in the form of invariants. Each invariant is defined in the context of a meta-class of the metamodel and specifies a name and a body. The body is an OCL expression that must evaluate to a Boolean result, indicating whether an instance of the meta-class satisfies the invariant or not. Execution-wise, the body of each invariant is evaluated for each instance of the meta-class and the results are stored in a set of <Element, Invariant, Boolean> triplets. Each triplet captures the Boolean result of the evaluation of an Invariant on a qualified Element. An exemplar OCL invariant for UML 1.4, requiring that abstract operations only belong to abstract classes, is shown in Listing 6.11.

Listing 6.11: OCL constraint on UML operations

context Operation
While in its current version OCL enables users to capture particularly complex invariants, it also demonstrates a number of shortcomings, as follows.

**Limited user feedback**

OCL does not support specifying meaningful messages that can be reported to the user in case an invariant is not satisfied for certain elements. Therefore, feedback to the user is limited to the name of the invariant and the instance(s) for which it failed. Weak support for proper feedback messages implies that the end users must be familiar with OCL so that they can comprehend the meaning of the failed invariant and locate the exact reason for the failure. This is a significant shortcoming as in practice only a very small number of end users are familiar with OCL.

**No support for warnings/critiques**

Contemporary software development environments typically produce two types of feedback when checking artefacts for consistency and correctness: errors and warnings. Errors indicate critical deficiencies that contradict basic principles and invalidate the developed artefacts. By contrast, warnings (or critiques) indicate non-critical issues that should nevertheless be addressed by the user. To enable users to address warnings in a priority-based manner, they are typically categorized into three levels of importance: High, Medium and Low (although other classifications are also possible).

Nevertheless, in OCL there is no such distinction between errors and warnings and consequently all reported issues are considered to be errors. This adds an additional burden to identifying and prioritizing issues of major importance, particularly within an extensive set of unsatisfied invariants in complex models.
No support for dependent constraints

Each OCL invariant is a self-contained unit that does not depend on other invariants. There are cases where this design decision is particularly restrictive. For instance consider the invariants $I_1$ and $I_2$ displayed in Listing 6.12. Both $I_1$ and $I_2$ are applicable on UML classes with $I_1$ requiring that: \textit{the name of a class must not be empty} and $I_2$ requiring that: \textit{the name of a class must start with a capital letter}. In the case of those two invariants, if $I_1$ is not satisfied for a particular UML class, evaluating $I_2$ on that class would be meaningless. In fact it would be worse than meaningless since it would consume time to evaluate and would also produce an extraneous error message to the user. In practice, to avoid the extraneous message, $I_2$ needs to replicate the body of $I_1$ using an \textit{if} expression (lines 2 and 5).

Listing 6.12: Conceptually related OCL constraints

```ocl
context Class
  inv I1 : self.name.size() > 0

inv I2 :
  if self.name.size > 0 then
    self.name.substring(0,1) =
    self.name.substring(0,1).toUpper()
  else
    true
  endif
```

Limited flexibility in context definition

As already discussed, in OCL invariants are defined in the context of meta-classes. While this achieves a reasonable partitioning of the model element space, there are cases where more fine-grained partitioning is required. For instance, consider the following scenario. Let $IA_{1..N}$, $IB_{1..M}$ be invariants applying to classes that are stereotyped as <<A>> and <<B>> respectively. Since OCL only supports partitioning the model element space using metaclasses, all $IA_{1..N}$, $IB_{1..M}$ must appear under the same context (i.e. Class). Moreover, each invariant must explicitly define that it addresses the one or
the other conceptual sub-partition. Therefore, each of $IA_{1..N}$ must limit its scope initially (using the `self.isA` expression) and then express the real body. In our example the simplest way to achieve this would be by combining a scope-limiting expression with the real invariant body using the `implies` clause as demonstrated in Listing 6.13.

Listing 6.13: Demonstration of OCL constraints with duplication

```ocm
context Class
  inv I1 : self.isA implies <real-invariant-body>
  inv I2 : self.isA implies <real-invariant-body>
  ...
  inv IN : self.isA implies <real-invariant-body>

def isA :
  let isA : Boolean =
    self.stereotype->exists(s|s.name = 'A')
```

Furthermore, if the `real` body of the invariant needs to assume that `self` is stereotyped with `<<A>>`, this technique is not applicable because OCL does not support lazy evaluation of Boolean clauses [57] and therefore although the first part of the expression (`self.isA`) may fail for some instances, the second part will still be evaluated thus producing runtime errors. In this case, an `if` expression must be used, further complicating the specified invariants.

**No support for repairing inconsistencies**

While OCL can be used for detecting inconsistencies, it provides no means for repairing them. The reason is that OCL has been designed as a side-effect free language and therefore lacks constructs for modifying models. Nevertheless, there are many cases where inconsistencies are trivial to resolve and users can benefit from semi-automatic repairing facilities.

This need has been long recognized in the related field of code development tools (e.g. Eclipse, Microsoft Visual Studio, NetBeans). In such tools, errors are not only identified but also context-aware actions are proposed to the user for automatically repairing them. This feature significantly
increases the usability of such tools and consequently enhances users’ productivity.

No support for inter-model constraints

As discussed in Section 2.4.1, OCL expressions (and therefore OCL constraints) can only be evaluated in the context of a single model at a time. Consequently, OCL cannot be used to express constraints that span across different models. In the context of a large-scale model driven engineering process that involves many different models (that potentially conform to different modelling languages) this limitation is particularly severe.

Following this discussion on the shortcomings of OCL for capturing structural constraints in modelling languages, the following sections present the abstract and concrete syntax of EVL as well as their execution semantics, and explain how they address the aforementioned limitations.

6.3.2 Abstract Syntax

In EVL, validation specifications are organized in modules (EvlModule). As illustrated in Figure 6.7, EvlModule extends EolLibraryModule which means that it can contain user-defined operations and import other EOL library modules and EVL modules. Apart from operations, an EVL module also contains a set of invariants grouped by the context they apply to, and a number of pre and post blocks.
Figure 6.7: Abstract Syntax of EVL
Context A context specifies the kind of instances on which the contained invariants will be evaluated. Each context can optionally define a guard which limits its applicability to a narrower subset of instances of its specified type. Thus, if the guard fails for a specific instance of the type, none of its contained invariants are evaluated.

Invariant As with OCL, each EVL invariant defines a name and a body (check). However, it can optionally also define a guard (defined in its abstract GuardedElement supertype) which further limits its applicability to a subset of the instances of the type defined by the embracing context. To achieve the requirement for detailed user feedback (Section 6.3.1), each invariant can optionally define a message as an ExpressionOrStatement-Block that should return a String providing a description of the reason(s) for which the constraint has failed on a particular element. To support semi-automatically fixing of elements on which invariants have failed (Section 6.3.1), an invariant can optionally define a number of fixes. Finally, as displayed in Figure 6.7 Invariant is an abstract class that is used as a super-class for the specific types Constraint and Critique. This is to address the issue of separation of errors and warnings/critiques (Section 6.3.1).

Guard Guards are used to limit the applicability of invariants (Section 6.3.1). This can be achieved at two levels. At the Context level it limits the applicability of all invariants of the context and at the Invariant level it limits the applicability of a specific invariant.

Fix A fix defines a title using an ExpressionOrStatementBlock instead of a static String to allow users to specify context-aware titles (e.g. Rename class customer to Customer instead of a generic Convert first letter to upper-case). Moreover, the do part is a statement block where the fixing functionality can be defined using EOL. The developer is responsible for ensuring that the actions contained in the fix actually repair the identified inconsistency.
**Constraint**  *Constraints* in EVL are used to capture critical errors that invalidate the model. As discussed above, *Constraint* is a sub-class of *Invariant* and therefore inherits all its features.

**Critique**  Unlike *Constraints*, *Critiques* are used to capture non-critical situations that do not invalidate the model, but should nevertheless be addressed by the user to enhance the quality of the model. This separation addresses the issue raised in Section 6.3.1. Moreover, to enable users to define different levels of importance in critiques, the *CritiqueLevel* enumeration supports a 3-level classification. Fixed-level classification has been preferred in EVL over infinite level classification (e.g. using Integer levels) since it is more common in development tools and easier to visualize.

**Pre and Post**  An EVL module can define a number of named *pre* and *post* blocks that contain EOL statements which are executed before and after evaluating the invariants respectively.

### 6.3.3 Concrete Syntax

Listings 6.14, 6.15 and 6.16 demonstrate the concrete syntax of the *context*, *invariant* and *fix* abstract syntax constructs discussed above.

**Listing 6.14: Concrete Syntax of an EVL context**

```plaintext
1 (context) <name> {  
2  (invariant)*  
3 }  
```

**Listing 6.15: Concrete Syntax of an EVL invariant**

```plaintext
1 (@lazy)?  
2 (constraint|critique) <name> {  
3  (guard {:expression}|{{statementBlock}})*  
4 }  
```
6.3.4 Execution Semantics

Having discussed the abstract and concrete syntaxes of EVL, this section provides an informal discussion of the execution semantics of the language. The execution of an EVL module is separated into four phases:

**Phase 1** Before any invariant is evaluated, the *pre* sections of the module are executed in the order in which they have been specified.

**Phase 2** For each *context*, the instances of the meta-class it defines are collected. For each instance, the *guard* of the *context* is evaluated. If the *guard* is satisfied, then for each non-lazy invariant contained in the context the invariant’s *guard* is also evaluated. If the *guard* of the invariant is satisfied, the *body* of the invariant is evaluated. In case the *body* evaluates to *false*, the *message* part of the rule is evaluated and the produced message
is added along with the instance, the invariant and the available fixes to the ValidationTrace.

The execution order of an EVL module follows a top-down depth-first scheme that respects the order in which the contexts and invariants appear in the module. However, the execution order can change in case one of the satisfies, satisfiesOne, satisfiesAll built-in operations, discussed in detail in the sequel, are called.

Phase 3  In this phase, the validation trace is examined for unsatisfied constraints and the user is presented with the message each one has produced. The user can then select one or more of the available fixes to be executed. Execution of fixes is performed in a transactional manner using the respective facilities provided by the model connectivity framework, as discussed in Section 5.1.2. This is to prevent runtime errors raised during the execution of a fix from compromising the validated model by leaving it in an inconsistent state.

Phase 4  When the user has performed all the necessary fixes or chooses to end Phase 3 explicitly, the post section of the module is executed. There, the user can perform tasks such as serializing the validation trace or producing a summary of the validation process results.

Capturing Dependencies Between Invariants

As discussed in Section 6.3.1, it is often the case that invariants conceptually depend on each other. To allow users to capture such dependencies, EVL provides the satisfies(invariant : String) : Boolean, satisfiesAll(invariants : Sequence(String)) : Boolean and satisfiesOne(invariants : Sequence(String)) : Boolean built-in operations. Using these operations, an invariant can specify in its guard other invariants which need to be satisfied for it to be meaningful to evaluate.

When one of these operations is invoked, if the required invariants (either lazy or non-lazy) have been evaluated for the instances on which the
operation is invoked, the engine will return their cached results; otherwise it will evaluate them and return their results.

6.3.5 Intra-Model Consistency Checking Example

This section presents a case study comparing EVL and OCL in the context of a common scenario. The purpose of the case study is to present readers with the concrete syntax of the language and demonstrate the benefits delivered by the additional constructs it facilitates.

Scenario: The Singleton Pattern

The singleton pattern is a widely used object oriented pattern. A singleton is a class for which exactly one instance is allowed [12]. In UML, a singleton is typically represented as a class which is stereotyped with a <<singleton>> stereotype and which also defines a static operation named getInstance() that returns the unique instance.

To ensure that all singletons have been modelled correctly in a UML model one needs to evaluate the following invariants on all classes that are stereotyped with the <<singleton>> stereotype:

- DefinesGetInstance : Each stereotyped class must define a getInstance() method
- GetInstanceIsStatic : The getInstance() method must be static
- GetInstanceReturnsSame : The return type of the getInstance() method must be the class itself

Obviously, invariants GetInstanceIsStatic and GetInstanceReturnsSame depend on DefinesGetInstance because if the singleton does not define a getInstance() operation, checking for the operation’s scope and return type is meaningless. Moreover, in case an invariant fails, there are corrective actions (fixes) that users may want to perform semi-automatically: e.g. for DefinesGetInstance, such an action would be to add the missing getInstance() operation, for GetInstanceIsStatic to change it to static and for GetInstanceRe-
*turnsSame* to set the return type to the class itself. In the following sections
OCL and EVL are used to express the three constraints and then the two
solutions are compared.

**Using OCL to Express the Invariants**

Listing 6.17 shows the aforementioned invariants implemented in OCL.

**Listing 6.17: OCL Module for Validating Singletons**

```ocl
class Foundation::Core

context Class

def isSingleton :
    let isSingleton : Boolean =
        self.stereotype->exists(s | s.name = 'singleton')

def getInstanceOperation :
    let getInstanceOperation : Operation =
        self.feature->select(f | f.oclIsTypeOf(Operation)
        and f.name = 'getInstance')->first().oclAsType(Operation)

inv DefinesGetInstanceOperation :
    if isSingleton
        then getInstanceOperation.isDefined
    else true
    endif

inv GetInstanceOperationIsStatic :
    if isSingleton then
        if getInstanceOperation.isDefined
            then getInstanceOperation.ownerScope = #classifier
        else false
        endif
    else
        true
    endif

inv GetOperationReturnsSame :
    if isSingleton then
```

159
if getInstanceOperation.isDefined then
  if getInstanceOperation.returnParameter.isDefined then
    getInstanceOperation.returnParameter.type = self
  else
    false
  endif
else
  false
endif
else
  true
endif

context Operation
def returnParameter :
  let returnParameter : Parameter =
  self.parameter->select(p|p.kind = #return)->first()
endpackage

By examining the OCL solution it can be observed that all invariants first check that the class is a singleton (lines 15, 21 and 31) by using the isSingleton derived property defined in line 5. If the isSingleton returns false, the invariants return true since returning false would cause them to fail for all non-singleton classes. This reveals an additional shortcoming of OCL: if a constraint returns true it may mean two different things: either that the instance satisfies the constraint or that the constraint is not applicable to the instance at all. In our view, this overloading reduces understandability.

By further studying the solution of Listing 6.17 it can be noticed that dependency between constraints is captured artificially using nested if expressions. For instance, both GetInstanceIsStatic and GetInstanceReturnsSame contain an if expression in lines 22 and 32 respectively, requiring that they recalculate the value of the getInstanceOperation defined in line 9, where they actually recalculate the result of the DefinesGetInstanceOperation invariant. As discussed in Section 6.3.1 this happens because OCL lacks constructs for capturing dependencies in a structured manner.
Using EVL to Express the Invariants

Listing 6.18 provides a solution for this problem expressed in EVL.

Listing 6.18: EVL Module for Validating Singletons

```
using EVL to Express the Invariants

Listing 6.18 provides a solution for this problem expressed in EVL.

Listing 6.18: EVL Module for Validating Singletons

```
self.getInstanceOperation.ownerScope := ScopeKind#sk_classifier; }

}

}

constraint GetInstanceReturnsSame { 
  guard : self.satisfies('DefinesGetInstance')
  check {
    var returnParameter : Parameter;
    returnParameter := self.getReturnParameter();
    return (returnParameter->isDefined()
      and returnParameter.type = self);
  }

  message: 'The getInstance() operation of singleton ' 
    + self.name + ' must return ' + self.name

  fix {
    title: 'Change return type to ' + self.name
    do {
      var returnParameter : Parameter;
      returnParameter := self.getReturnParameter();

      -- If the operation does not have a return parameter
      -- create one
      if (not returnParameter.isDefined()){
        returnParameter := Parameter.newInstance();
        returnParameter.kind := ParameterDirectionKind#pdk_return;
        returnParameter.behavioralFeature :=
          self.getInstanceOperation();
      }

      -- Set the correct return type
      returnParameter.type := self;
    }
  }

}

operation Class getInstanceOperation() : Operation {

}
The Singleton context defines that the invariants it contains will be evaluated on instances of the UML Class type. Moreover, its guard defines that they will be evaluated only on classes that are stereotyped with the singleton stereotype. Therefore, unlike the OCL solution of Listing 6.17, invariants contained in this context do not need to check individually that the instances on which they are evaluated are singletons.

Constraint DefinesGetInstance defines no guard which means that it will be evaluated for all the instances of the context. In its check part, the constraint examines if the class defines an operation named getInstance() by invoking the getGetInstanceOperation() operation. If this fails, it proposes a fix that adds the missing operation to the class.

Constraint GetInstanceIsStatic defines a guard which states that for the constraint to be evaluated on an instance, the instance must first satisfy the DefinesGetInstance constraint. If it doesn’t, it is not evaluated at all. In its check part it examines that the getInstance() operation is static. Note that here the constraint needs not check that the getInstance() operation is defined again since this is assumed by the DefinesGetInstance constraint on which it depends. If the constraint fails for an instance, the fix part can be invoked to change the scope of the getInstance() operation to static.

Constraint GetInstanceReturnsSame checks that the return type of the getInstance() operation is the singleton itself. Similarly to the GetInstanceIsStatic constraint, it defines that to be evaluated the DefinesGetInstance constraint must be satisfied. If it fails for a particular instance, the fix part can be invoked. In the fix part, if the operation defines a return parameter of incorrect type, its type is changed and if it does not define a return pa-
rameter at all, the parameter is created and added to the parameters of the operation.

By observing the two solutions the OCL solution resembles the concept of defensive programming, where conditions are embedded in supplier code, while the EVL one is closer to the design by contract [134] approach where conditions are explicitly checked in guards.

This case study has demonstrated that the additional constructs provided by EVL can reduce repetition significantly and thus enable specification of more concise constraints. Moreover, in case a constraint is not satisfied for a particular instance, the user is provided with a meaningful context-aware message and with automated facilities (fixes) for repairing the inconsistency.

### 6.3.6 Inter-Model Consistency Checking Example

In the previous example, EVL was used to check the internal consistency of a single UML model. By contrast, this example demonstrates using EVL to detect and repair occurrences of incompleteness and contradiction between two different models. In this example the simplified ProcessLang metamodel, which captures information about hierarchical processes, is used. To add performance information in a separate aspect ProcessPerformanceLang metamodel is also defined. The metamodels are displayed in Figures 6.8 and 6.9 respectively.

There are two constraints that need to be defined and evaluated in this example: that each Process in a process model (PM) has a corresponding ProcessPerformance in the process performance model (PPM), and that the maxAcceptableTime of a process does not exceed the sum of the maxAcceptableTimes of its children. This is achieved with the PerformanceIsDefined and the PerformanceIsValid EVL constraints displayed in Listing 6.19.

Listing 6.19: Exemplar EVL module containing a cross-model constraint

```euml
context PM!Process {

    constraint PerformanceIsDefined {

```
check {
  var processPerformances :=
    PPM!ProcessPerformance.
    allInstances.select(pt|pt.process = self);
  return processPerformances.size() = 1;
}

message {
  var prefix : String;
  if (processPerformances.size() = 1) {
    prefix := 'More than one performance info';
  }
else {
    prefix := 'No performance info';
}

return prefix + ' found for process ' + self.name;

fix {
    title: 'Set the performance of ' + self.name

    do {
        for (p in processPerformances.clone()) {
            delete p;
        }

        var maxAcceptableTime : Integer;
        maxAcceptableTime := UserInput.promptInteger('maxAcceptableTime', 0);
        var p :
            new PPM!ProcessPerformance;
        p.maxAcceptableTime := maxAcceptableTime;
        p.process := self;
    }
}

constraint PerformanceIsValid {
    guard : self.satisfies('PerformanceIsDefined')
    and self.children.forAll
        (c|c.satisfies('PerformanceIsDefined'))

    check {
        var sum : Integer;
        sum := self.children.
            collect(c|c.getMaxAcceptableTime())
            .sum().asInteger();
        return self.getMaxAcceptableTime() >= sum;
    }
}
message : 'Process ' + self.name +
' has a smaller maxAcceptableTime '
+ 'than the sum of its children'

fix {
   title : 'Increase maxAcceptableTime to ' + sum
   do {
      self.setMaxAcceptableTime(sum);
   }
}

operation PM!Process getMaxAcceptableTime()
: Integer {
   return PPM!ProcessPerformance.
   allInstances.selectOne(pt|pt.process=self)
   .maxAcceptableTime;
}

operation PM!Process setMaxAcceptableTime
(time : Integer) {
   PPM!ProcessPerformance.allInstances.
   selectOne(pt|pt.process=self).maxAcceptableTime :=
   time;
}

In line 5 the check part of the PerformanceIsDefined constraint calculates the instances of ProcessPerformance in the ProcessPerformanceModel that have their process reference set to the currently examined Process (accessible via the self built-in variable) and stores it in the processPerformances variable. If exactly one ProcessPerformance is defined for the Process, the constraint is satisfied. Otherwise, the message part of the constraint, in line 13, is evaluated and an appropriate error message is displayed to the user.

Note that the processPerformances variable defined in the check part is also used from within the message part of the constraint. As discussed in
EVL provides this feature to reduce the need for duplicate calculations as our experience has shown that the message for a failed constraint often needs to utilize side-information collected in the check part.

To repair the inconsistency, the user can invoke the fix defined in line 25 that will delete any existing ProcessPerformance instances and create a new one with a user-defined maxAcceptableTime obtained using the User-Input.promptInteger() statement of line 33.

Unlike the PerformanceIsDefined constraint, the PerformanceIsValid constraint, line 43 defines a guard part (line 45). As discussed in [18], the guard part of a constraint is used to further limit the applicability of the constraint beyond the simple type check performed in the containing context. In this rule, the validity of the maxAcceptableTime of a Process needs to be checked only if one has been defined in the ProcessPerformanceModel. Therefore, the guard part of the constraint specifies that this constraint is only applicable to Processes where, both they and they children, satisfy the PerformanceIsDefined constraint.

The check part of the constraint retrieves the maxAcceptableTime of the process and that of its children and compares them. As the Process itself does not define performance information, retrieval of the value of the maxAcceptableTime of the respective ProcessPerformance object is implemented using the user-defined getMaxAcceptableTime() operation that is defined in line 72. In case the constraint is not satisfied, the user can invoke the fix defined in line 61 to repair the inconsistency by setting the maxAcceptableTime of the process to the sum calculated in line 51. As discussed earlier, the fix parts of EVL invariants do not in any way guarantee that they do fix the problem they target or that in their effort to fix one problem they do not create another problem; this is left to the user. For instance, in this particular example, changing the maxAcceptableTime of a process through a fix block may render its parent process invalid.

To demonstrate the evaluation of these constraints two exemplar models that conform to the ProcessLang and ProcessPerformanceLang metamodels are used. A visual representation of the models is displayed in Figure 6.10.
Figure 6.10: Exemplar Process and ProcessPerformance models

Figure 6.11: Screenshot of the validation view reporting the identified inconsistencies

problems which are reported to the user via the view displayed in Figure 6.11. Indeed by examining the two models of Figure 6.10 it becomes apparent that there is no ProcessPerformance linked to the Verify PIN process and also that the maxAcceptableTime of Complete Transaction (5000) is less than the sum of the maxAcceptableTimes of its children (2000 + 3500).

6.3.7 Summary

This section has provided a detailed discussion on the EVL model-validation language which conceptually (as opposed to technically) extends OCL. EVL
provides a number of features such as support for detailed user feedback, constraint dependency management, semi-automatic transactional inconsistency resolution and (as it is based on EOL) access to multiple models of diverse metamodels and technologies.
As discussed in Section 2.4.2, there are two types of model-to-model transformations: mapping and update transformations. Mapping transformations typically transform a source model into a set of target models expressed in (potentially) different modelling languages by creating zero or more model elements in the target models for each model element of the source model. By contrast, update transformations perform in-place modifications in the source model itself. They can be further classified into two subcategories: transformations in the small and in the large. Update transformations in the large apply to sets of model elements calculated using well-defined rules in a batch manner. An example of this category of transformations is a transformation that automatically adds accessor and mutator operations for all attributes in a UML model. On the other hand, update transformations in the small are applied in a user-driven manner on model elements that have been explicitly selected by the user. An example of this kind of transformations is a transformation that renames a user-specified UML class and all its incoming associations consistently.

In Epsilon, mapping transformations can be specified using ETL as discussed in Section 6.2 and update transformations in the large can be implemented either using the model modification features of EOL or using an ETL transformation in which the source and target models are the same model. By contrast, update transformations in the small cannot be effectively addressed by any of the languages presented so far.

The following section discusses the importance of update transformations in the small and motivates the definition of a task-specific language (Epsilon Wizard Language (EWL)) that provides tailored and effective support for defining and executing update transformations on models of diverse metamodels.

### 6.4.1 Motivation

Constructing and refactoring models is undoubtedly a mentally intensive process. However, during modelling, recurring patterns of model update
activities typically appear. As an example, when renaming a class in a UML class diagram, the user also needs to manually update the names of association ends that link to the renamed class. Thus, when renaming a class from *Chapter* to *Section*, all association ends that point to the class and are named *chapter* or *chapters* should be also renamed to *section* and *sections* respectively. As another example, when a modeller needs to refactor a UML class into a singleton [42], they need to go through a number of well-defined, but trivial, steps such as attaching a stereotype (*<< singleton >>*), defining a static *instance* attribute and adding a static *getInstance()* method that returns the unique instance of the singleton.

It is generally accepted that performing repetitive tasks manually is both counter-productive and error-prone [135]. On the other hand, failing to complete such tasks correctly and precisely compromises the consistency, and thus the quality, of the models. In Model Driven Engineering, this is particularly important since models are increasingly used to automatically produce (parts of) working systems.

**Automating the Construction and Refactoring Process**

Contemporary modelling tools provide built-in transformations (*wizards*) for automating common repetitive tasks. However, according to the architecture of the designed system and the specific problem domain, additional repetitive tasks typically appear, which cannot be addressed by the preconceived built-in wizards of a modelling tool. To address the automation problem in its general case, users must be able to easily define update transformations (*wizards*) that are tailored to their specific needs.

To an extent, this can be achieved via the extensible architecture that state-of-the-art modelling tools often provide and which enables users to add functionality to the tool via scripts or application code using the implementation language of the tool. Nevertheless, as discussed in [3], the majority of modelling tools provide an API through which they expose an edited model, which requires significant effort to learn and use. Also, since each API is proprietary, such scripts and extensions are not portable to other
tools. Finally, API scripting languages and third-generation languages such as Java and C++ are not particularly suitable for model navigation and modification. Furthermore, as discussed in Section 2.4.2, existing languages for mapping transformations, such as QVT and ATL, cannot be used as-is for this purpose, because these languages have been designed to operate in a batch manner without human involvement in the process. By contrast, as discussed in the next section, the task of constructing and refactoring models is inherently user-driven and as such, it demonstrates a different set of requirements.

6.4.2 Update Transformations in the Small

Update transformations are actions that automatically create, update or delete model elements based on a selection of existing elements in the model and information obtained otherwise (e.g. through user input), in a user-driven fashion. In this section such actions are referred to as wizards instead of rules to reduce confusion between them and rules of mapping transformation languages. In the following sections the desirable characteristics of wizards are elaborated informally.

Structure of Wizards

In its simplest form, a wizard only needs to define the actions it will perform when it is applied to a selection of model elements. The structure of such a wizard that transforms a UML class into a singleton is shown using pseudo-code in Listing 6.20

Listing 6.20: The simplest form of a wizard for refactoring a class into a singleton

```plaintext
do :
  attach the singleton stereotype
  create the instance attribute
  create the getInstance method
```

Listing 6.20: The simplest form of a wizard for refactoring a class into a singleton
Since not all wizards apply to all types of elements in the model, each wizard needs to specify the types of elements to which it applies. For example, the wizard of Listing 6.20, which automatically transforms a class into a singleton, applies only when the selected model element is a class. The simplest approach to ensuring that the wizard will only be applied on classes is to enclose its body in an `if` condition as shown in Listing 6.21.

Listing 6.21: The wizard of Listing 6.20 enhanced with an `if` condition

```java
do :
    if (selected element is a class) {
        attach the singleton stereotype
        create the instance attribute
        create the getInstance method
    }
```

A more modular approach is to separate this condition from the body of the wizard. This is shown in Listing 6.22 where the condition of the wizard is specified as a separate `guard` stating that the wizard applies only to elements of type Class. The latter is preferable since it enables filtering out wizards that are not applicable to the current selection of elements by evaluating only their `guard` parts and rejecting those that return `false`. Thus, at any time, the user can be provided with only the wizards that are applicable to the current selection of elements. Filtering out irrelevant wizards reduces confusion and enhances usability, particularly as the list of specified wizards grows.

Listing 6.22: The wizard of Listing 6.21 with an explicit `guard` instead of the `if` condition

```java
guard : selected element is a class
do :
    attach the singleton stereotype
    create the instance attribute
    create the getInstance method
```

To enhance usability, a wizard also needs to define a short human-readable description of its functionality. To achieve this, another field named `title` has been added. There are two options for defining the title of a wizard:
the first is to use a static string and the second to use a dynamic expression. The latter is preferable since it enables definition of context-aware titles.

Listing 6.23: The wizard of Listing 6.22 enhanced with a title part

```
guard : selected element is a class
title : Convert class <class-name> into a singleton
do :
  attach the singleton stereotype
  create the instance attribute
  create the getInstance method
```

Capabilities of Wizards

The guard and title parts of a wizard need to be expressed using a language that provides model querying and navigation facilities. Moreover, the do part also requires model modification capabilities to implement the transformation. To achieve complex transformations, it is essential that the user can provide additional information. For instance, to implement a wizard that addresses the class renaming scenario discussed in Section 6.4.1, the information provided by the selected class does not suffice; the user must also provide the new name of the class. Therefore, EWL must also provide mechanisms for capturing user input.

6.4.3 Abstract Syntax

Since EWL is built atop Epsilon, its abstract and concrete syntax need only to define the concepts that are relevant to the task it addresses; they can reuse lower-level constructs from EOL. A graphical overview of the abstract syntax of the language is provided in Figure 6.12.

The basic concept of the EWL abstract syntax is a Wizard. A wizard defines a name, a guard part, a title part and a do part. Wizards are organized in Modules. The name of a wizard acts as an identifier and must be unique in the context of a module. The guard and title parts of a wizard are of type ExpressionOrStatementBlock, inherited from EOL. An ExpressionOrStatementBlock is either a single EOL expression or a block of EOL
statements that include one or more `return` statements. This construct allows users to express simple declarative calculations as single expressions and complex calculations as blocks of imperative statements. The usefulness of this construct is further discussed in the examples presented in Section 6.4.6. Finally, the `do` part of the wizard is a block of EOL statements that specify the effects of the wizard when applied to a compatible selection of model elements.
6.4.4 Concrete Syntax

Listing 6.24 presents the concrete syntax of EWL wizards.

Listing 6.24: Concrete syntax of EWL wizards

```ewl
wizard <name> {  
  (guard (:expression)|{{statementBlock}})?  
  (title (:expression)|{{statementBlock}})?  
  do {  
    statementBlock  
  }  
}
```

6.4.5 Execution Semantics

The process of executing EWL wizards is inherently user-driven and as such it depends on the environment in which they are used. In general, each time the selection of model elements changes (i.e. the user selects or deselects a model element in the modelling tool), the guards of all wizards are evaluated. If the guard of a wizard is satisfied, the title part is also evaluated and the wizard is added to a list of applicable wizards. Then, the user can select a wizard and execute its do part to perform the intended transformation.

In EWL, variables defined and initialized in the guard part of the wizard can be accessed both by the title and the do parts. In this way, results of calculations performed in the guard part can be re-used, instead of re-calculated in the subsequent parts. The practicality of this approach is discussed in more detail in the examples that follow. Also, the execution of the do part of each wizard is performed in a transactional mode by exploiting the transaction capabilities of the underlying model connectivity framework, as discussed in Section 5.1.2 so that possible logical errors in the do part of a wizard do not leave the edited model in an inconsistent state.

6.4.6 Examples

This section presents three concrete examples of EWL wizards for refactoring UML 1.4 models. The aim of this section is not to provide complete
implementations that address all the sub-cases of each scenario but to pro-
vide enhanced understanding of the concrete syntax, the features and the
capabilities of EWL to the reader. Moreover, it should be stressed again that
although the examples in this section are based on UML models, by build-
ing on Epsilon, EWL can be used to capture wizards for diverse modelling
languages and technologies.

Converting a Class into a Singleton

The singleton pattern [42] is applied when there is a class for which only
one instance can exist at a time. In terms of UML, a singleton is a class
stereotyped with the << singleton >> stereotype, and it defines a static at-
tribute named instance which holds the value of the unique instance. It also
defines a static getInstance() operation that returns that unique instance.
Wizard ClassToSingleton, presented in Listing 6.25 simplifies the process of
converting a class into a singleton by adding the proper stereotype, attribute
and operation to it automatically.

Listing 6.25: Implementation of the ClassToSingleton Wizard

```plaintext
wizard ClassToSingleton {
  -- The wizard applies when a class is selected
  guard : self.isTypeOf(Class)

  title : 'Convert ' + self.name + ' to a singleton'

  do {
    -- Create the getInstance() operation
    var gi : new Operation;
    gi.owner := self;
    gi.name := 'getInstance';
    gi.visibility := VisibilityKind#vk_public;
    gi.ownerScope := ScopeKind#sk_classifier;

    -- Create the return parameter of the operation
    var ret : new Parameter;
    ret.type := self;
  }
}
```
The *guard* part of the wizard specifies that it is only applicable when the selection is a single UML class. The *title* part specifies a context-aware title that informs the user of the functionality of the wizard and the *do* part implements the functionality by adding the *getInstance* operation (lines 10-14),
the *instance* attribute (lines 23-28) and the «*singleton*» stereotype (line 31).

The stereotype is added via a call to the *attachStereotype()* operation. Attaching a stereotype is a very common action when refactoring UML models, particularly where UML profiles are involved, and therefore to avoid duplication, this reusable operation that checks for an existing stereotype, creates it if it does not already exists, and attaches it to the model element on which it is invoked has been specified.

An extended version of this wizard could also check for existing association ends that link to the class and for which the upper-bound of their multiplicity is greater than one and either disallow the wizard from executing on such classes (in the *guard* part) or update the upper-bound of their multiplicities to one (in the *do* part). However, the aim of this section is not to implement complete wizards that address all sub-cases but to provide a better understanding of the concrete syntax and the features of EWL. This principle also applies to the examples presented in the sequel.

**Renaming a Class**

The most widely used convention for naming attributes and association ends of a given class is to use a lower-case version of the name of the class as the name of the attribute or the association end. For instance, the two ends of a one-to-many association that links classes *Book* and *Chapter* are most likely to be named *book* and *chapters* respectively. When renaming a class (e.g. from *Chapter* to *Section*) the user must then manually traverse the model to find all attributes and association ends of this type and update their names (i.e. from *chapter* or *bookChapter* to *section* and *bookSection* respectively). This can be a daunting process especially in the context of large models. Wizard *RenameClass* presented in Listing 6.26 automates this process.

**Listing 6.26: Implementation of the RenameClass Wizard**

```java
1 wizard RenameClass {  
2 }
```

180
-- The wizard applies when a Class is selected

\[
guard : self.isKindOf(Class)
\]

\[
title : 'Rename class ' + self.name
\]

\[
do { 
  var newName : String;

  -- Prompt the user for the new name of the class
  newName := UserInput.prompt('New name for class ' + self.name);
  if (newName.isDefined()) {
    var affectedElements : Sequence;

    -- Collect the AssociationEnds and Attributes
    -- that are affected by the rename
    affectedElements.addAll(
      AssociationEnd.allInstances.select(ae|ae.participant=self));
    affectedElements.addAll(
      Attribute.allInstances.select(a|a.type = self));

    var oldNameToLower : String;
    oldNameToLower := self.name.firstToLowerCase();
    var newNameToLower : String;
    newNameToLower := newName.firstToLowerCase();

    -- Update the names of the affected AssociationEnds
    -- and Attributes
    for (ae in affectedElements) {
      ae.replaceInName(oldNameToLower, newNameToLower);
      ae.replaceInName(self.name, newName);
    }
    self.name := newName;
  }
}
\]

-- Renames the ModelElement on which it is invoked

\[
operation ModelElement replaceInName
\]

\[
(oldString : String, newString : String) {
\]

181
As with the ClassToSingleton wizard, the guard part of RenameClass specifies that the wizard is applicable only when the selection is a simple class and the title provides a context-aware description of the functionality of the wizard.

As discussed in Section 6.4.2 the information provided by the selected class itself does not suffice in the case of renaming since the new name of the class is not specified anywhere in the existing model. In EWL, and in all languages that build on EOL, user input can be obtained using the built-in UserInput facility. Thus, in line 12 the user is prompted for the new name of the class using the UserInput.prompt() operation. Then, all the association ends and attributes that refer to the class are collected in the affectedElements sequence (lines 14-21). Using the replaceInName operation (lines 31 and 32), the name of each one is examined for a substring of the upper-case or the lower-case version of the old name of the class. In case the check returns true, the user is prompted to confirm (line 48) that the feature needs to be renamed. This further highlights the importance of user input for implementing update transformations with fine-grained user control.
Moving Model Elements into a Different Package

A common refactoring when modelling in UML is to move model elements, particularly Classes, between different packages. When moving a pair of classes from one package to another, the associations that connect them must also be moved in the target package. To automate this process, Listing 6.27 presents the MoveToPackage wizard.

Listing 6.27: Implementation of the MoveToPackage Wizard

```plaintext
wizard MoveToPackage {
  -- The wizard applies when a Collection of
  -- elements, including at least one Package
  -- is selected
  guard {
    var moveTo : Package;
    if (self.isKindOf(Collection)) {
      moveTo := self.select(e|e.isKindOf(Package)).last();
    }
    return moveTo.isDefined();
  }

  title: 'Move ' + (self.size()-1) + ' elements to ' + moveTo.name

  do {
    -- Move the selected Model Elements to the
    -- target package
    for (me in self.excluding(moveTo)) {
      me.namespace := moveTo;
    }

    -- Move the Associations connecting any
    -- selected Classes to the target package
    for (a in Association.allInstances) {
      if (a.connection.forAll(c|self.includes(c.participant))){
        a.namespace := moveTo;
      }
    }
  }
}
```


The wizard applies when more than one element is selected and at least one of the elements is a Package. If more than one package is selected, the last one is considered as the target package to which the rest of the selected elements will be moved. This is specified in the guard part of the wizard.

To reduce user confusion in identifying the package to which the elements will be moved, the name of the target package appears in the title of the wizard. This example shows the importance of the decision to express the title as a dynamically calculated expression (as opposed to a static string). It is worth noting that in the title part of the wizard (line 14), the moveTo variable declared in the guard (line 7) is referenced. Through experimenting with a number of wizards, it has been noticed that in complex wizards repeated calculations need to be performed in the guard, title and do parts of the wizard. To eliminate this duplication, the scope of variables defined in the guard part has been extended so that they are also accessible from the title and do part of the wizard.

6.4.7 Summary

This section has presented the Epsilon Wizard Language (EWL), a language for specifying and executing update transformations in the small on models of diverse metamodels. EWL provides a textual concrete syntax tailored to the task and features such as dynamically calculated wizard titles, transactional execution of the do parts of wizards and user interaction.

6.5 The Epsilon Merging Language (EML)

The aim of EML is to contribute model merging capabilities to Epsilon. More specifically, EML can be used to merge an arbitrary number of input models of potentially diverse metamodels and modelling technologies. This section provides a discussion on the motivation for implementing EML, its abstract and concrete syntax, as well as its execution semantics. It also provides two examples of merging homogeneous and heterogeneous models.
6.5.1 Motivation

A mechanism that enables automatically merging models on a set of established correspondences has a number of applications in a model driven engineering process. For instance, it can be used to unify two complementary, but potentially overlapping, models that describe different views of the same system. In another scenario, it can be used to merge a core model with an aspect model (potentially conforming to different metamodels), as discussed in [136] where a core Platform Independent Model (PIM) is merged with a Platform Definition Model (PDM), that contributes platform-specific aspects, into a Platform Specific Model (PSM).

Phases of Model Merging

Existing research [101, 102] has demonstrated that model merging can be decomposed into four distinct phases: comparison, conformance checking, merging and reconciliation (or restructuring).

Comparison Phase In the comparison phase, correspondences between equivalent elements of the source models are identified, so that such elements are not propagated in duplicate in the merged model. Several approaches to comparison, ranging from manual to fully automatic, have been discussed in Section 2.4.3.

Conformance Checking Phase In this phase, elements that have been identified as matching in the previous phase are examined for conformance with each other. The purpose of this phase is to identify potential conflicts that would render merging infeasible. The majority of proposed approaches, such as [118], address conformance checking of models complying with the same metamodel.

Merging Phase Several approaches have been proposed for the merging phase. In [101, 119], graph-based algorithms for merging models of the same metamodel are proposed. In [118], an interactive process for merging of UML
2.0 models is presented. There are at least two weaknesses in the methods proposed so far. First, they only address the issue of merging models of the same metamodel, and some of them address a specific metamodel indeed. Second, they use an inflexible merging algorithm and do not provide means for extending or customizing its logic.

**Reconciliation and Restructuring Phase** After the merging phase, the target model may contain inconsistencies that need fixing. In the final step of the process, such inconsistencies are removed and the model is polished to acquire its final form. Although the need for a reconciliation phase is discussed in [102, 119], in the related literature the subject is not explicitly targeted.

**Relationship between Model Merging and Model Transformation**

A merging operation is a transformation in a general sense, since it transforms some input (source models) into some output (target models). However, as discussed throughout this section, a model merging facility has special requirements (support for comparison, conformance checking and merging pairs of input elements) that are not required for typical one-to-one or one-to-many transformations [87] and are therefore not supported by contemporary model transformation languages.

**6.5.2 Realizing a Model Merging Process with Epsilon**

The first two steps of the process described above can be realized with existing languages provided by Epsilon. As discussed in Section 6.1, the comparison step can be realized with the Epsilon Comparison Language (ECL). Following that, the Epsilon Validation Language (EVL) can be used to validate the identified correspondences using the match trace calculated by ECL. The Epsilon Merging Language (EML) presented below provides support for the last two steps of the process (merging and reconciliation/restructuring).
6.5.3 Abstract Syntax

In EML, merging specifications are organized in modules (EmlModule). As displayed in Figure 6.13, EmlModule inherits from EtlModule.
Figure 6.13: The Abstract Syntax of EML
By extending \textit{EtlModule}, an EML module can contain a number of transformation rules and user-defined operations. An EML module can also contain one or more merge rules as well as a set of pre and post named statement blocks.

Each merge rule defines a name, a left, a right, and one or more target parameters. It can also extend one or more other merge rules and be defined as having one or more of the following properties: abstract, greedy, lazy and primary.

6.5.4 Concrete Syntax

Listing 6.28 demonstrates the concrete syntax of EML merge-rules.

Listing 6.28: Concrete syntax of an EML merge-rule

\begin{verbatim}
(@abstract)?
(@lazy)?
(@primary)?
(@greedy)?
rule <name>
  merge <leftParameter>
  with <rightParameter>
  into (<targetParameter>{, <targetParameter>}*)?
  {extends <ruleName>{, <ruleName>}*)? {

  statementBlock
}
\end{verbatim}

6.5.5 Execution Semantics

Rule and Block Overriding

An EML module can import a number of other EML and ETL modules. In this case, the importing EML module inherits all the rules and pre/post blocks specified in the modules it imports (recursively). If the module specifies a rule or a pre/post block with the same name, the local rule/block overrides the imported one respectively.
Rule Scheduling

When an EML module is executed, the pre blocks are executed in the order in which they have been defined.

Following that, for each match of the established matchTrace the applicable non-abstract, non-lazy merge rules are executed. When all matches have been merged, the transformation rules of the module are executed on all applicable elements - that have not been merged - in the models.

Finally, after all rules have been applied, the post blocks of the module are executed. In post blocks, EOL statements can be used to restructure and reconcile the merged model in an imperative manner in case this is deemed necessary.

Rule Applicability

By default, for a merge-rule to apply to a match, the left and right elements of the match must have a type-of relationship with the leftParameter and rightParameter of the rule respectively. This can be relaxed to a kind-of relationship by specifying that the merge rule is greedy (using the @greedy annotation in terms of concrete syntax).

Source Elements Resolution

As with model transformation, in model merging it is often required to resolve the counterparts of an element of a source model into the target models. In EML, this is achieved by overloading the semantics of the equivalents() and equivalent() operations defined by ETL. In EML, in addition to inspecting the transformation trace and invoking any applicable transformation rules, the equivalents() operation also examines the mergeTrace (displayed in Figure 6.14) that stores the results of the application of merge-rules and invokes any applicable (both lazy and non-lazy) rules.

Similarly to ETL, the order of the results of the equivalents() operation respects the order of the (merge or transform) rules that have produced them. An exception to that occurs if one of the rules has been declared
as primary, in which case its results are prepended to the list of elements returned by equivalent.

6.5.6 Homogeneous Model Merging Example

In this scenario, two models conforming to the Graph metamodel need to be merged. The first step is to compare the two graphs using the ECL module of Listing 6.29.

Listing 6.29: ECL module for comparing two instances of the Graph metamodel

```ecl
1 rule MatchNodes
```

Figure 6.14: The EML runtime
match l : Left!Node
with r : Right!Node {
    compare : l.label = r.label
}

rule MatchEdges
match l : Left!Edge
with r : Right!Edge {
    compare : l.source.matches(r.source)
    and l.target.matches(r.target)
}

rule MatchGraphs
match l : Left!Graph
with r : Right!Graph {
    compare : true
}

The MatchNodes rule in line 1 defines that two nodes match if they have the same label. The MatchEdges rule in line 8 specifies that two edges match if both their source and target nodes match (regardless of whether the labels of the edges match or not as it is assumed that there can not be two distinct edges between the same nodes). Finally, since only one instance of Graph is expected to be in each model, the MatchGraphs rule in line 16 returns true for any pair of Graphs.

Having established the necessary correspondences between matching elements of the two models, the EML specification of listing 6.30.

Listing 6.30: EML module for merging two instances of the Graph meta-model on the correspondences identified in Listing 6.29

import 'Graphs.etl';

rule MergeGraphs

\[2^2\]Both assumptions can be checked using EVL before matching/merging takes place but this is out of the scope of this example
In line 3, the MergeGraphs merge rule specifies that two matching Graphs (l and r) are to be merged into one Graph t in the target model that has as a
label, the concatenation of the labels of the two input graphs separated using 'and'. The Nodes merge rule in line 22 specifies that two matching Nodes are merged into a single Node in the target model. The label of the merged node is derived by concatenating the c (for common) static string with the label of the source Node from the left model. Similarly, the MergeEdges rule specifies that two matching Edges are merged into a single Edge in the target model. The source and target nodes of the merged Edge are set to the equivalents (::=) of the source and target nodes of the edge from the left model.

To reduce duplication, the MergeNodes and MergeEdges rules extend the abstract MergeGraphElements rule specified in line 13 which assigns the graph property of the graph element to the equivalent of the left graph.

The rules displayed in Listing 6.30 address only the matching elements of the two models. To also copy the elements for which no equivalent has been found in the opposite model, the EML module imports the ETL module of Listing 6.31.

Listing 6.31: The Graphs.etl ETL transformation module

```plaintext
rule TransformGraph
  transform s : Source!Graph
  to t : Target!Graph {
    t.label := s.label;
  }

@abstract
rule TransformGraphElement
  transform s : Source!GraphElement
  to t : Target!GraphElement {
    t.graph ::= s.graph;
  }

rule TransformNode
  transform s : Source!Node
  to t : Target!Node
```

194
The rules of the ETL module apply to model elements of both the Left and the Right model as both have been aliased as Source. Of special interest is the TransformNode rule in line 17 that specifies that non-matching nodes in the two input models will be transformed into nodes in the target model the labels of which will be a concatenation of their input graph and the label of their counterparts in the input models.

Executing the ECL and EML modules of Listings 6.29 and 6.30 on the exemplar models displayed in Figures 6.15 and 6.16 creates the target model of Figure 6.17.
6.6 The Epsilon Generation Language (EGL)

In addition to the languages above, the Epsilon Generation Language (EGL) is a model-to-text language that also builds atop the infrastructure provided by Epsilon. However, the contribution of the author to the design and development of EGL was limited to the construction of an initial prototype and therefore no credit is claimed for the current state of the language and its tool-support in this report. A detailed discussion of the language can be found in [28].
6.7 Implementing a New Task-Specific Language

Although Epsilon already provides languages for a wide range of model management tasks, additional tasks that could benefit from the convenient syntax and dedicated semantics of a task-specific language are likely to be identified in the future. Thus, this section distills the experiences obtained through the construction of existing task-specific languages to provide guidance on how to identify a task for which a dedicated language can be beneficial and develop the respective task-specific language for it atop the infrastructure provided by Epsilon.

6.7.1 Identifying the need for a new language

The first step of the process of constructing a new task-specific language is to identify a specific task for which a dedicated language is more appropriate than the general-purpose EOL. Typically, recurring syntactic and semantic patterns that emerge when attempting to implement the task using EOL indicate that a new task-specific language may be useful.

For example, before the introduction of the Epsilon Comparison Language, pure EOL was being used to perform model comparison. A simple comparison specification that establishes name-based matches between classes/attributes and tables/columns between two OO and DB models respectively using EOL is demonstrated in Listing 6.32.

Two patterns can be readily detected by inspecting the EOL code in Listing 6.32. First, explicit variables (matchingCT, matchingAT) are defined to capture the matching elements (class-table and attribute-column) identified during the comparison process. Also, to check all elements of one type (classes against tables and attributes against columns) repeated for statements are used in lines 3–4 and 7–8. By contrast, Listing 6.33 which is specified using the task-specific ECL language does not include such low-level information. Instead it defines only the types of elements that need to be compared and the criteria on which comparison must performed and leaves the mundane tasks of scheduling and maintaining the match trace to the execution engine.
Listing 6.32: Comparing an OO model with a DB model using EOL

```eol
var matchingCT : Sequence;
var matchingAC : Sequence;
for (c in OO!Class.allInstances) {
    for (t in DB!Table.allInstances) {
        if (t.name = c.name) {
            matchingCT.add(Sequence{c, t});
            for (att in c.attributes) {
                for (col in t.columns) {
                    if (att.name = c.name) {
                        matchingAC.add(Sequence{att, col});
                    }
                }
            }
        }
    }
}
```

Listing 6.33: Comparing an OO model with a DB model using ECL

```ecl
rule ClassTable
match c : OO!Class
with t : DB!Table {
    compare : c.name = t.name
}

rule AttributeColumn
match a : OO!Attribute
with c : DB!Column {
    compare : a.name = c.name and a.class.matches(c.table)
}
```

### 6.7.2 Eliciting higher-level constructs from recurring patterns

Once recurring patterns, such as those discussed above, have been identified, the next step of the process is to derive higher level constructs from them.
For instance, in the previous example, the nested for loops and the explicit trace variable declaration and population have been replaced by task-specific match rules.

Introducing higher-level involves defining its abstract and concrete syntax as well as its connection points with the underlying infrastructure. For example, in the case of ECL, the types of match rules are EOL model element types, the guard and check parts of a rule are EOL expressions or statements blocks and the pre and post blocks as well as the do part of each rule are blocks of EOL statements.

6.7.3 Implement Execution Semantics and Scheduling

Once higher-level constructs (e.g. task-specific rules) have been identified and specified, their execution semantics and scheduling must be implemented similarly to what has been done for existing languages. Development of existing languages has demonstrated that task-specific constructs often need to provide more than one modes of execution (e.g. the lazy and greedy modes of ETL transformation rules discussed in Section 6.2.5).

A lightweight way to easily provide new execution modes and semantics for rules and user-defined operations without modifying the syntax of the language and introducing new keywords that may conflict with existing code, is through the annotations mechanism provided by EOL (see Section 5.2.3). This approach has been adopted for the definition of a small unit-testing language (EUnit), which is discussed in detail in [17].

6.7.4 Overriding Semantics

In certain cases, it is useful to modify the semantics of certain constructs in EOL to meet the purposes of the task-specific language. An example of such a modification occurs in EVL where – as discussed in Section 6.3.4 – the scope of the variables defined in guard expression/block is extended so that variables can be reused in the context of non-nested blocks such as the title, and check parts of the invariant. Another example of overriding the semantics of EOL is the implementation of the special assignment operator
6.8 Chapter Summary

This chapter has presented a number of task specific languages that build atop the infrastructure discussed in Chapter 5 and which address the tasks of model comparison (ECL), model-to-model transformation (ETL), model merging (EML), model validation (EVL) and in-place model transformation (EWL). For each language, the abstract and concrete syntaxes have been presented along with an informal discussion of the execution semantics, and a number of representative concrete examples of the use of the language. Furthermore, in Section 6.7, based on the experience acquired by designing and developing the above languages, guidance for identifying additional tasks that can benefit from task-specific languages and implementing them atop the infrastructure provided by Epsilon is provided.

Having discussed in detail the structure and semantics of existing task-specific languages and the process of implementing a new language, the next chapter presents a workflow mechanism that provides seamless integration between different task-specific languages to enable composition of individual tasks into complex model management workflows.
Chapter 7

Orchestration Workflow

The previous chapter has provided a detailed discussion on a number of task-specific languages, each one addressing an individual model management task. However, as discussed in Section 2.5, in practice, model management tasks are seldom carried out in isolation; instead, they are often combined together to form complex workflows. Therefore, of similar importance to the existence of individual task-specific management languages is the provision of a mechanism that enables developers to compose modular and reusable tasks into complex automated processes. In a broader context, to facilitate implementation of seamless workflows, an appropriate MDE workflow mechanism should also support mainstream development tasks such as file management, version control management, source code compilation and invocation of external programs and services.

7.1 Motivation

As a motivating example, an exemplar workflow that consists of both MDD tasks (1-4, 6) and mainstream software development tasks (5, 7) is displayed below.

1. Load a UML model
2. Validate it
3. Transform it into a Database Schema model

4. Generate Java code from the UML model

5. Compile the Java code

6. Generate SQL code from the Database model

7. Deploy the SQL code in a Database Management System (DBMS)

In the above workflow, if the validation step (2) fails, the entire process should be aborted and the identified errors should be reported to the user. This example demonstrates that to be of practical use, a task orchestration framework needs to be able to coordinate both model management and mainstream development tasks and provide mechanisms for establishing dependencies between different tasks.

This chapter presents such a framework for orchestrating modular model management tasks implemented using languages of the Epsilon platform. As the problem of task coordination is common in software development, many technical solutions have been already proposed and are widely used by software practitioners. In this context, designing a new general-purpose workflow management solution was deemed inappropriate. Therefore, the task orchestration solution discussed here has been designed as an extension to the robust and widely used ANT framework. A brief overview of ANT as well as a discussion on the choice to design the orchestration workflow of Epsilon atop it is provided below.

### 7.2 The ANT Tool

ANT, named so because it is *a little thing that can be used to build big things*, is a robust and widely-used framework for composing automated workflows from small reusable activities. The most important advantages of ANT, compared to traditional build tools such as *gnumake*, is that it is platform independent and easily extensible. Platform independence is achieved by building atop Java, and extensibility is realized through
a lightweight binding mechanism that enables developers to contribute custom tasks using well defined interfaces and extension points.

Although a number of tools with functionality similar to ANT exist in the Java community, only Maven [140] is currently of comparable magnitude in terms of user-basis size and robustness. Outlining the discussion provided in [141], ANT is considered to be easier to learn and to enable low-level control, while Maven is considered to provide a more elaborate task organization scheme. Nevertheless, the two frameworks are significantly similar and the ANT technical solution discussed in this chapter can easily be ported to work with the latter.

This section provides a brief discussion of the structure and concrete syntax of ANT workflows, as well as the extensibility mechanisms that ANT provides to enable users contribute custom tasks.

### 7.2.1 Structure

In ANT, each workflow is captured as a *project*. A simplified illustration of the structure of an ANT project is displayed in Figure 7.1. Each ANT project consists of a number of *targets*. The one specified as the *default* is executed automatically when the project is executed. Each *target* contains a number of *tasks* and *depends* on other targets that must be executed before it. An ANT task is responsible for a distinct activity and can either succeed or fail. Exemplar activities implemented by ANT tasks include file system management, compiler invocation, version management and remote artefact deployment.

### 7.2.2 Concrete Syntax

In terms of concrete syntax, ANT provides an XML-based syntax. In Listing 7.1, an exemplar ANT project that compiles a set of Java files is illustrated. The project contains one target (*main*) which is also set to be the *default* target. The *main* target contains one *javac* task that specifies attributes such as *sdir*, *destdir* and *classpath*, which define that the Java compiler will compile a set of Java files contained into the *src* directory into classes.
that should be placed in the build directory using dependencies.jar as an external library.

Listing 7.1: Compiling Java classes using the javac task

```xml
<project default="main">
    <target name="main"/>
    <javac srcdir="${src}"
        destdir="${build}"
        classpath="dependencies.jar"
        debug="on"
        source="1.4"/>
</target>
```
7.2.3 Extending ANT

Binding between the XML tags that describe the tasks and the actual implementations of the tasks is achieved through a light-weight mechanism at two levels. First, the tag (in the example of Listing 7.1, javac) is resolved to a Java class that extends the org.apache.ant.Task abstract class (in the case of javac, the class is org.apache.tools.ant.taskdefs.Javac) via a configuration file. Then, the attributes of the tasks (e.g. srcdir) are set using the reflective features that Java provides. Finally, the execute() method of the task is invoked to perform the actual job.

This lightweight and straightforward way of defining tasks has rendered ANT particularly popular in the Java development community and currently there is a large number of tasks contributed by ANT users [142], ranging from invoking tools such as code generators and XSLT processors, to emulating logical control flow structures such as if conditions and while loops. The AMMA platform [143] also provides integration of model driven engineering tools such as TCS [124] and ATL [83] with ANT.

ANT also supports more advanced features including nested XML elements and filesets, however, providing a complete discussion is beyond the scope of this paper. For a definitive guide to ANT readers can refer to [138].

7.3 Integration Challenges

A simple approach to extending ANT with support for model management tasks would be to implement one standalone task for each language in Epsilon. However, such an approach demonstrates a number of integration and performance shortcomings which are discussed below.

Since models are typically serialized in the file system, before a task is executed, the models it needs to access/modify must be parsed and loaded in memory. In the absence of a more elaborate framework, each model management task would have to take responsibility for loading and storing
the models it operates on. Also, in most workflows, more than one task operates on the same models sequentially, and needlessly loading/storing the same models many times in the context of the same workflow is an expensive operation both time and memory-wise, particularly as the size of models increases.

Another weakness of this primitive approach is limited inter-task communication. In the absence of a communication framework that allows model management tasks to exchange information with each other, it is often the case that many tasks end up performing the same (potentially expensive) queries on models. By contrast, an inter-task communication framework would enable time and resource intensive calculations to be performed once and their results to be communicated to all interested subsequent tasks.

Having discussed ANT, Epsilon and the challenges their integration poses, the following sections present the design of a solution that enables developers to invoke model management tasks in the context of ANT workflows. The solution consists of a core framework that addresses the challenges discussed in Section 7.3, a set of specific tasks, each of which implements a distinct model management activity, and a set of tasks that enable developers to initiate and manage transactions on models using the respective facilities provided by the model connectivity layer discussed in Section 5.1.2.

7.4 Framework Design and Core Tasks

The role of the core framework, illustrated in Figure 7.2, is to provide model loading and storing facilities as well as runtime communication facilities to the individual model management tasks that build atop it. This section provides a detailed discussion of the components it consists of.
Figure 7.2: Core Framework
7.4.1 The EpsilonTask task

An ANT task can access the project in which it is contained by invoking the `Task.getProject()` method. To facilitate sharing of arbitrary information between tasks, ANT projects provide two convenience methods, namely `addReference(String key, Object ref)` and `getReference(String key) : Object`. The former is used to add key-value pairs, which are then accessible using the latter from other tasks of the project.

To avoid loading models multiple times and to enable on-the-fly management of models from different Epsilon modules without needing to store and re-load the models after each task, a reference to a project-wide model repository has been added to the current ANT project using the `addReference` method discussed above. In this way, all the subclasses of the abstract `EpsilonTask` can invoke the `getProjectRepository()` method to access the project model repository.

Also, to support a variable sharing mechanism that enables inter-task communication, the same technique has been employed; a shared context, accessible by all Epsilon tasks via the `getProjectContext()` method, has been added. Through this mechanism, model management tasks can export variables to the project context (e.g. traces or lists containing results of expensive queries) which other tasks can then reuse.

`EpsilonTask` also specifies a `profile` attribute that defines if the execution of the task must be profiled using the profiling features provided by Epsilon. Profiling is a particularly important aspect of workflow execution, especially where model management languages are involved. The main reason is that model management languages tend to provide convenient features which can however be computationally expensive (such as the `allInstances()` EOL built-in feature that returns all the instances of a specific metaclass in the model) and when used more often than really needed, can significantly degrade the overall performance.
7.4.2 Model Loading Task

The LoadModelTask (*epsilon.loadModel*) loads a model from an arbitrary location (e.g. file-system, database) and adds it to the project repository so that subsequent Epsilon tasks can query or modify it. Since Epsilon supports many modelling technologies (e.g. EMF, MDR, XML), the LoadModelTask defines only three generic attributes. The name attribute specifies the name of the model in the project repository. The type attribute specifies the modelling technology with which the model is captured and is used to resolve
the technology-specific model loading functionality. Finally, the aliases attribute defines a comma-separated list of alternative names by which the model can be accessed in the model repository.

The rest of the information needed to load a model is implementation-specific and is therefore provided through parameter nested elements, each one defining a pair of name-value attributes. As an example, a task for loading an EMF model that has a file-based ECore metamodel is displayed in Listing 7.2.

Listing 7.2: Loading an EMF model using the epsilon.loadModel task

```xml
<epsilon.loadModel name="Tree1" type="EMF">
  <parameter name="modelFile" value="TreeInstance.ecore"/>
  <parameter name="metamodelFile" path="Tree.ecore"/>
  <parameter name="isMetamodelFileBased" value="true"/>
  <parameter name="readOnlyLoad" value="true"/>
</epsilon.loadModel>
```

### 7.4.3 Model Storing Task

The StoreModelTask (epsilon.storeModel) is used to store a model residing in the project repository. The StoreModelTask defines two attributes. The name attribute specifies the name of the model to be stored and the target attribute specifies the location where the model will be stored. The target attribute is optional and if it is not defined, the model is stored in the location from which it was originally loaded.

### 7.4.4 Model Disposal Tasks

When a model is no longer required by tasks of the workflow, it can be disposed using the epsilon.disposeModel task. The task provides the ref attribute that defines the name of the model to be disposed. Also, the attribute-less epsilon.disposeModels task is provided that disposes all the models in the project model repository. This task is typically invoked when the model management part of the workflow has finished.
The workflow leverages the model-transaction services provided by the model connectivity framework of Epsilon by providing three tasks for managing transactions in the context of workflows.

### 7.4.5 The StartTransaction Task

The `epsilon.startTransaction` task defines a `name` attribute that identifies the transaction. It also optionally defines a comma-separated list of model names (`models`) that the transaction will manage. If the `models` attribute is not specified, the transaction involves all the models contained in the common project model repository.

### 7.4.6 The CommitTransaction and RollbackTransaction Tasks

The `epsilon.commitTransaction` and `epsilon.rollbackTransaction` tasks define a `name` attribute through which the transaction to be committed/rolled-back is located in the project’s active transactions. If several active transactions with the same name exist the more recent one is selected.

The example of Listing 7.3 demonstrates an exemplar usage of the `epsilon.startTransaction` and `epsilon.rollbackTransaction` tasks. In this example, two empty models Tree1 and Tree2 are loaded in lines 1,2. Then, the EOL task of line 4 queries the models and prints the number of instances of the `Tree` metaclass in each one of them (which is 0 for both). Then, in line 13 a transaction named T1 is started on model Tree1. The EOL task of line 15 creates a new instance of `Tree` in both Tree1 and Tree2 and prints the number of instances of `Tree` in the two models (which is 1 for both models). Then, in line 26 the T1 transaction is rolled-back and any changes done in its context to model Tree1 (but not Tree2) are undone. Therefore, the EOL task of line 28 which prints the number of instances of `Tree` in both models, prints 0 for Tree1 but 1 for Tree2.

Listing 7.3: Exemplar usage of the `epsilon.startTransaction` and `epsilon.rollbackTransaction` tasks

```xml
<epsilon.loadModel name="Tree1" type="EMF">...<epsilon.loadModel>
<epsilon.loadModel name="Tree2" type="EMF">...<epsilon.loadModel>
```
7.4.7 The Abstract Executable Module Task

This task is the base of all the model management tasks presented in Section 7.5. Its aim is to encapsulate the commonalities of Epsilon tasks in order to reduce duplication among them. As already discussed, in Epsilon,
specifications of model management tasks are organized in executable modules. While modules can be stored anywhere, in the case of the workflow it is assumed that they are either stored as separate files in the file-system or they are provided inline within the workflow. Thus, this abstract task defines an src attribute that specifies the path of the source file in which the Epsilon module is stored, but also supports inline specification of the source of the module. The two alternatives are demonstrated in Listings 7.4 and 7.5 respectively.

Listing 7.4: External Module Specification

```xml
<project default="main">
  <target name="main">
    <epsilon.eol src="HelloWorld.eol"/>
  </target>
</project>
```

Listing 7.5: Inline Module Specification

```xml
<project default="main">
  <target name="main">
    <epsilon.eol>
      <![CDATA[
        'Hello world'.println();
      ]]> 
    </epsilon.eol>
  </target>
</project>
```

The task also defines the following nested elements:

0..n model nested elements Through the model nested elements, each task can define which of the models, loaded in the project repository it needs to access. Each model element defines three attributes. The ref attribute specifies the name of the model that the task needs to access, the as attribute defines the name by which the model will be accessible in the context of the task, and the aliases defines a comma-delimited sequence of aliases for the model in the context of the task.
0..n parameter nested elements The parameter nested elements enable users to communicate String parameters to tasks. Each parameter element defines a name and a value attribute. Before executing the module, each parameter element is transformed into a String variable with the respective name and value which is then made accessible to the module.

0..n exports nested elements To facilitate low-level integration between different Epsilon tasks, each task can export a number of variables to the project context, so that subsequent tasks can access them later. Each export nested element defines the three attributes. The ref attribute specifies the name of the variable to be exported, the as string attribute defines the name by which the variable is stored in the project context and the optional boolean attribute specifies whether the variable is mandatory. If optional is set to false and the module does not specify such a variable, an ANT BuildException is raised.

0..n uses nested elements The uses nested elements enable tasks to import variables exported by previous Epsilon tasks. Each use element supports three attributes. The ref attribute specifies the name of the variable to be used. If there is no variable with this name in the project context, the ANT project properties are queried. This enables Epsilon modules to access ANT parameters (e.g. provided using command-line arguments). The as attribute specifies the name by which the variable is accessible in the context of the task. Finally, the optional boolean parameter specifies if the variable must exist in the project context.

To better illustrate the runtime communication mechanism, a minimal example is provided in Listings 7.6 - 7.8. In Listing 7.6 Exporter.eol defines a String variable named x and assigns a value to it. The workflow of Listing 7.8 specifies that after executing Exporter.eol, it must export a variable named x with the new name y to the project context. Finally, it defines that before executing User.eol (Listing 7.7), it must query the project context for a variable named y and in case this is available, add the variable to the module’s context and then execute it. Thus, the result of executing the
workflow is *Some String* printed in the output console.

Listing 7.6: Source code of the Exporter.eol module

```
var x : String := 'Some string';
```

Listing 7.7: Source code of the User.eol module

```
z.println();
```

Listing 7.8: ANT Workflow connecting modules 7.6 and 7.7 using the epsilon.eol task

```
<epsilon.eol src="Exporter.eol">
  <exports ref="x" as="y"/>
</epsilon.eol>

<epsilon.eol src="User.eol">
  <uses ref="y" as="z"/>
</epsilon.eol>
```

### 7.5 Model Management Tasks

Having discussed the core framework, this section presents the model management tasks that have been implemented atop it, using languages of the Epsilon platform.

#### 7.5.1 Generic Model Management Task

The `epsilon.eol` task executes an EOL module, defined using the `src` attribute on the models that are specified using the `model` nested elements.

#### 7.5.2 Model Validation Task

The `epsilon.evl` task executes an EVL module, defined using the `src` attribute on the models that are specified using the `model` nested elements. In addition to the attributes defined by the ExecutableModuleTask, this task also provides the following attributes:
Figure 7.4: Model Management Tasks

- **failOnErrors**: Errors are the results of unsatisfied constraints. Setting the value of this attribute to `true` (default is `false`) causes a `BuildException` to be raised if one or more errors are identified during the validation process.

- **failOnWarnings**: Similarly to errors, warnings are the results of unsatisfied critiques. Setting the value of this attribute to `true` (default is also `false`) causes a `BuildException` to be raised if one or more warnings are identified during the validation process.

- **exportConstraintTrace**: This attribute enables developers to export the internal constraint trace constructed during model validation to the project context so that it can be later accessed by other tasks - which could for example attempt to automatically repair the identified
inconsistencies.

7.5.3 Model-to-Model Transformation Task

The \texttt{epsilon.etl} task executes an ETL module, defined using the \textit{src} attribute to transform between the models that are specified using the \textit{model} nested elements. In addition to the attributes defined by the ExecutableModuleTask, this task also provides the \textit{exportTransformationTrace} attribute that enables the developer to export the internal transformation trace to the project context. In this way this trace can be reused by subsequent tasks; for example another task can serialize it in the form of a separate traceability model.

7.5.4 Model Comparison Task

The \texttt{epsilon.ecl} task executes an ECL module, defined using the \textit{src} attribute to establish matches between elements of the models that are specified using the \textit{model} nested elements. In addition to the attributes defined by the ExecutableModuleTask, this task also provides the \textit{exportMatchTrace} attribute that enables users to export the match-trace calculated during the comparison to the project context so that subsequent tasks can reuse it. For example, as discussed in the sequel, an EML model merging task can use it as a means of identifying correspondences on which to perform merging. In another example, the match-trace can be stored by a subsequent EOL task in the form of a stand-alone weaving model.

7.5.5 Model Merging Task

The \texttt{epsilon.eml} task executes an EML module, defined using the \textit{src} attribute on the models that are specified using the \textit{model} nested elements. In addition to the attributes defined by the ExecutableModuleTask, this task also provides the following attributes:

- \textit{useMatchTrace} : As discussed in \textsection{6.5} to merge a set of models, an EML module needs an established match-trace between elements of
the models. The useMatchTrace attribute enables the EML task to use a match-trace exported by a preceding ECL task (using its exportMatchTrace attribute).

- exportMergeTrace, exportTransformationTrace: Similarly to ETL, through these attributes an EML task can export the internal traces calculated during merging for subsequent tasks to use.

### 7.5.6 Model-to-Text Transformation Task

To support model to text transformations, EglTask (epsilon.egl) task is provided that executes an Epsilon Generation Language (EGL) module\(^1\). In addition to the attributes defined by ExecutableModuleTask, EglTask also defines a target attribute that defines where the path of the file where the generated text will be stored.

### 7.5.7 Adding a new Model Management Task

As discussed in Section 6.7 additional task-specific languages are likely to be needed in the future for tasks that are not effectively supported by existing task-specific languages. In addition to designing and implementing the syntax and execution semantics of a new language, it is also important to provide integration with the workflow – if the nature of the language permits execution within a workflow. As a counter-example, no workflow task has been provided for EWL since its execution semantics is predominately user-driven and as such, it makes little sense to execute EWL in the context of an automated workflow.

To implement support for a new task-specific language to the workflow, a new extension of the abstract ExecutableModuleTask needs to be provided (similarly to what has been done for existing task-specific languages). By extending ExecutableModuleTask, the task is automatically provided with access to the essential features of the workflow such as the shared model repository, and runtime context. Additional configuration options for the

---

\(^{1}\) As discussed in Section 6.6 EGL has been built atop Epsilon with a minimal contribution of the author
task need to specified as new ANT attributes and/or nested elements, similarly to what has been done for the tasks presented in Sections 7.5.1, 7.5.6.

7.6 Chapter Summary

This chapter has presented the detailed design of an ANT-based framework for integrating and orchestrating mainstream software development tasks with model management tasks implemented using model management languages presented in Chapters 5 and 6 into seamless workflows. In Section 7.4 the core framework that provides features such centralized model loading/storing facilities, a shared model repository and a mechanism through which individual tasks can communicate at runtime has been illustrated. Then, Section 7.5 has provided a discussion on the integration of the task specific languages discussed in Chapter 6 with the framework and also provided guidance for adding support for additional languages that are likely to be developed in the future atop Epsilon.
Chapter 8

Reference Implementation

In Chapters 5–7, the design of the proposed platform, the languages built atop it and the orchestration framework that enables composition of individual tasks into coherent workflows have been discussed in detail. To build confidence in the soundness of the proposed approach and the feasibility of the design, and to enable external users, particularly partners of the ModelWare and ModelPlex EU projects, to use the proposed approach in practice and provide useful feedback, a reference implementation of the infrastructure, task-specific languages and orchestration workflow has been constructed. The reference implementation provides execution engines for the languages and the orchestration workflow and end-user development tools that enable users to compose and execute Epsilon programs and workflows.

Java [144] has been selected as the language of choice for the implementation of Epsilon for a number of reasons; Java is a robust object-oriented language that can be executed in a wide range of software and hardware platforms, there are high-quality open-source development tools (such as the Eclipse JDT [58]) for composing, debugging and executing Java programs, and finally the most robust and widely used modelling frameworks (EMF and MDR) that Epsilon needed to interface with have been also implemented using Java.

This chapter presents an overview of the reference implementation and
Section 8.1 provides an overview of the mechanisms that are used to parse and execute Epsilon programs. Section 8.2 enumerates the modelling technologies that are supported by the reference implementation. Sections 8.3 and 8.4 provide an overview of the Eclipse-based development tools through which users can compose and execute Epsilon programs and workflows. Finally, Section 8.5 discusses the evolution and availability of the reference implementation in time, from its initial versions to the time of writing this report (May 2008).

8.1 Execution Engines

The reference implementation provides an execution engine for each language of the platform. As displayed in Figure 8.1, each execution engine consists of four parts: lexer, parser, model builder and interpreter. The lexer and parser components are responsible for parsing textual programs into abstract syntax trees (ASTs) [145]. Although the lexer and parser for a language can be hand-crafted, they are typically generated from a higher-level artefact that describes the grammar of the language. In this reference implementation, the robust and widely-used ANTLR [146] parser generator tool has been used for this purpose. Thus, for each language in the platform, an ANTLR grammar has been specified and from this, the respective lexers and parsers have been automatically generated. The main reason for selecting ANTLR among other tools with similar functionality was that ANTLR (in versions 2.x) supports grammar-level inheritance and this feature made it possible to realize the inheritance relationship that exists between task-specific languages and EOL without duplicating the EOL grammar in each of the task-specific languages grammars.

After parsing the textual program into an AST, the high-level logical constructs of the language (e.g. transformation-rules in ETL, operations in EOL etc.) are extracted from the AST into the logical model from the model
Figure 8.1: Overview of the Execution Engine of a typical Epsilon language builder component. By contrast, lower-level constructs such as statements and expressions are maintained in the form of ASTs. Finally, the interpreter is responsible for executing the hybrid model.

The reason for this hybrid structure of the logical model (mixture of language-specific logical constructs and ASTs) and the use of interpreters instead of compilers is that the main concern in this exploratory reference implementation was modifiability and extensibility. Therefore, in prospect of the anticipated changes to the syntax and semantics of the languages that would - and did indeed - arise during the lifecycle of the project, it was decided that the most lightweight and flexible possible approach should be chosen.

However, the flexibility achieved with such a light-weight approach negatively affects attributes such as performance and usability. For example, a complete logical model for each language would arguably enhance performance and also enable static analysis which is necessary to implement usability enhancing features such as context aware code completion and detection of logical errors (e.g. accessing undefined properties). As discussed later in Section 10.4, since the syntax and semantics of the core and task-specific languages of the platform have been now stabilised, and thus they are not expected to change significantly, future versions of the implementation shall adopt the later approach to deliver improved performance and usability.
8.2 Modelling Technologies

The reference implementation contains EMC drivers (implementations of the \textit{IModel} interface discussed in Section \ref{sec:reflection}) for the following modelling technologies.

- MOF 2.0, XMI 2.x using the Eclipse Modeling Framework (EMF)
- MOF 1.4, XMI 1.x using the Netbeans Metadata Repository (MDR)
- Plain XML
- Z, LaTeX (contributed by Wei Liu as part of an MSc thesis \cite{147} that the author supervised)

The uniform access mechanism implemented by EMC enables model management programs expressed using languages in Epsilon to access and modify any number of models in these technologies in any combination (e.g. transform a MOF 1.4-based UML model to a Z model, compare a plain XML model with a MOF 2.0-based DSL model).

8.3 Eclipse-based Development Tools

To be of practical use, each language in Epsilon is supported by a set of development tools built atop the Eclipse platform \cite{58}. This section provides a brief discussion on Eclipse and the Epsilon development tools implemented atop it.

8.3.1 The Eclipse Platform

Eclipse is an open-source software development framework supported by a large and active development and user community. As illustrated in Figure \ref{fig:eclipse摺紙}, Eclipse adopts a modular architecture which consists of a small core and well-defined extension mechanisms that enable developers to contribute additional functionality in the form of \textit{plugins} (which can be grouped in the form of \textit{features} and \textit{products}).
A wide range of extensible and widely-used tools have been developed atop Eclipse including editors and launchers for many programming languages (Java, C, PHP, Ruby etc.), management tools for source code repositories such as CVS and SVN and modelling tools such as the Eclipse Modeling Framework (EMF) and the Graphical Modeling Framework (GMF).

Due to its extensible nature, Eclipse enables users to implement supporting tools for new programming languages that integrate well with the rest of the platform with relatively little effort. Moreover, Eclipse is the platform of choice for MDE tools as the majority of contemporary MDE languages and frameworks provide Eclipse-based development tools.
8.3.2 Editors and Outline Views

To enable users to compose programs in languages of the platform, languagespecific editors aware of the concrete syntax of each language are required. Moreover, to enable users locate and correct syntactical errors, in-place markers that can highlight the lines that contain the error(s) are particularly useful. To maximize reuse, an abstract AbstractModuleEditor class has been introduced (that extends the built-in TextEditor Eclipse editor), which provides basic syntax highlighting services for keywords, comments, strings and numbers that apply to all Epsilon languages. Editors of individual languages (e.g. EclEditor and EtlEditor) extend this abstract editor to provide their specific keywords (e.g. compare and transform respectively). Similarly, a ModuleContentOutlinePage has been introduced that implements a basic outline view providing services such as linking with editors and sorting (Figure 8.3 point 5). To specialize for a task-specific language, little customization (specifying icons and labels) is required. As an example, in Figure 8.3, the editor (1) and outline view (4), that provides a compact view of the transform-rules of an ETL module, are illustrated. Points (2) and (3) show how design-time errors are presented with the use of markers both in the editor and the problems view of Eclipse.
Figure 8.3: ETL Editor and Outline View
8.3.3 Launch Configuration Interface

Since languages in Epsilon are about managing models, a major sub-task when creating a launch configuration is to define the models it operates on. To achieve reuse and uniformity, separate tabs and dialogs have been implemented, each being able to configure models of a particular technology (e.g. EMF, MDR). These tabs are reused in the launch configuration interfaces of all Epsilon languages. Figure 8.4 illustrates the MDR and EMF model configuration tabs (1) and the dialog through which the name and extent (2), as well as the locations of the model and metamodel file (3), of an MDR model can be specified.
8.3.4 The Epsilon Console

To provide the user with feedback at run-time, a console dedicated to Epsilon languages has been implemented. Users can send textual messages to this console via the built-in \texttt{print()} and \texttt{println()} EOL operations. Moreover, as demonstrated in Figure 8.5, when a run-time error occurs during the execution of a module, an appropriate message (1) and a hyperlink (2) are displayed in the console to inform the user about the error. Clicking on the hyperlink navigates the user to the actual source of the error (3).

8.4 Workflow Implementation

As discussed in Chapter 7, the workflow mechanism for orchestrating individual Epsilon tasks into composite workflows has been designed atop the infrastructure provided by the ANT framework. Eclipse already provides out-of-the-box integration with ANT including extensible editors, views and launch configurations which have been extended to enable users to compose
Figure 8.6: Task content assistance dialog

and execute workflows that involve instances of the Epsilon tasks presented in Sections 7.4 and 7.5. Figure 8.6 demonstrates a content-assistance dialog that enables user to select one of the Epsilon tasks and figure 8.7 demonstrates the content assistance dialog that demonstrates the valid attributes for an ETL task.

8.5 Timeline and Availability

Initially, the source code and documentation of the implementation was maintained locally. Since May 2006 new binary versions of the execution engines and the Eclipse-based development tools have been released regularly through the author’s departmental website. In November 2006, Epsilon was invited to become a component of the Eclipse GMT project. The Eclipse Generative Modeling Tools (GMT) project is the official research incubator project of the top-level Eclipse Modelling Project and hosts a small num-
ber of selected components (8 at the time of joining and 11 in May 2008). In January 2007 the source code was moved to the Eclipse CVS (concurrent version control) servers where it is since then accessible to the general public. Also, updated binary versions are regularly released as standalone archives and using the Eclipse Update Manager. Detailed instructions on how to obtain the source code and binaries are available in [148] and [149] respectively.

8.6 Chapter Summary

This chapter has provided an overview of the reference implementation that enables users to compose and execute programs expressed using languages built atop Epsilon. Through this chapter, noteworthy implementation decisions such as the choice to implement a light-weight runtime and development tools atop the Eclipse platform have been discussed, and an outline of
the timeline of the implementation process has been presented. This chapter concludes the design and implementation of Epsilon. The next chapter evaluates the validity of the hypothesis using a number of criteria, including a complex case study which has been realized using the reference implementation discussed here.
Chapter 9

Evaluation

In this thesis a novel approach to the construction of uniform, consistent, and interoperable task-specific languages for model management has been proposed. This chapter provides an evaluation of the research hypothesis stated in Section 3.2, the process that was followed to investigate it, and of the obtained research results, from several viewpoints. In Section 9.1 an extensive case study that requires use of a number of languages developed in this thesis is presented. The aim of the case study is to demonstrate the engineering qualities and practicality of the proposed solution. The description of the case study was set up by the organizers of the Model Driven Tools Implementers Forum (MDD-TIF) [150] in 2007, and besides the implementation using Epsilon, a number of additional implementations using state-of-the-art MDE tools and languages were presented during the workshop [150].

In Section 9.2 the impact of the work on the Model Driven Engineering community is assessed. Several factors are taken into consideration for this purpose; namely, the publications produced as a result of this work and their visibility, the acceptance and visibility of the proposed solution in the MDE community, and the exploitation of the research outcome in the context of the ModelWare and ModelPlex research projects.

Section 9.3 demonstrates the benefits, in terms of reuse, delivered by the layered nature of the proposed architecture. Reuse is evaluated under two
perspectives. From the language engineering perspective, reuse is measured in terms of the size of the code required to implement the concrete syntax, execution engine and development tools of each language in the platform. From the user perspective, reuse is evaluated in terms of the ability of the user to reuse functionality in different model management operations and programs.

Section 9.4 provides a comparison of Epsilon with other platforms of similar scope, and Section 9.5 discusses the limitations and shortcomings of this work; most notably the absence of mathematically expressed formal semantics. Section 9.6 evaluates the qualities of the reference implementation and provides directions for its evolution.

9.1 Case Study

This section presents a case study that uses combinations of languages of the Epsilon platform to implement a number of model management tasks. The scenario and requirements of the case study have been defined by the organizers of the Model Driven Development Tools Implementers Forum (MDD-TIF) [150] to enable a comparison of several different MDE tools using a common basis.

9.1.1 Scenario

The following case study description has been taken as-is from the MDD-TIF web-site [150].

Abstract

Digital television allows interactive content to accompany standard broadcasts. The development of bespoke interactive content is expensive. You are to design a system that will allow the non-technical producers of television programmes to build interactive content from a set of high-level building blocks. In effect, you will create a modelling language (metamodel, rules and symbols), a tool supporting it, and a generator.
Figure 9.1: Playing along with Test the Nation

**Background**

Digital television is becoming increasingly popular in the UK. In addition to providing higher quality video and increased channel capacity, it allows interactive content to accompany standard broadcasts. Interactive applications have been used to enhance traditional broadcasts in many ways:

- Viewers can play along with quizzes (Figure 9.1).
- Viewers can choose different camera angles during sporting events.
- Viewers can take part in discussions and comment on events through message boards.
- Viewers can remind themselves of the important developments in a drama’s plot (Figure 9.2).

**Problem**

Currently, each interactive application is bespoke. This greatly limits the number of programmes that can be accompanied by interactive content, as the applications are expensive to develop. The resulting UI is also different for different programmes, which is confusing to users. You are to design a system that will allow non-technical producers to build applications to accompany their programmes.
Figure 9.2: Viewing developments in The Murder Game

The application will sit on the right hand side of the screen (Figure 9.3), and display one of the following pieces of content:

- A page of text, to be used for news stories, background information etc.
- A multiple choice vote, for example *Man of the match* in a football match.
- A menu that allows the user to view items of content, including sub-menus.

The basic on-screen layout and navigational structure of the application has been defined by the user experience department, and producers are not able to change it.

**Use cases**

The following set of use cases emerged during discussions with the producers. They are ordered by their value to the producers, with the use case giving the most value first.
Case 1  A producer would like to build an application for the soccer world cup finals, listing the teams and information about each, and allowing viewers to vote for the most likely winner.

Case 2  As Case 1, but the teams should be listed by group (e.g. four teams in three groups). As the teams are known before their division into groups by lots, it should be possible to define the content for the teams first, and quickly add the structure of the groups, so that the application can be running for users as soon as possible during the program that broadcasts the division by lots.

Case 3  A producer would like to use a page of text to provide analysis of the recent events in a rugby match. A journalist with a laptop will need to change the text on the page throughout the match. The user interface used by the journalist should not allow him to change the structure of the whole application, only edit the text.

Case 4  A producer has decided that the wording of a particular text page was better before the last set of changes, and would like to revert it.
Interactive application architecture

Designing interactive applications requires relatively detailed knowledge of the standards for each platform. For this reason, you should concentrate on the system used by producers to define the application content, and its interfaces to black box components that build the actual application. To define these interfaces, you will need some knowledge of the basic architecture of interactive applications. The following crash-course should suffice. Digital televisions contain a basic operating system, and a set of libraries providing functions to display text and graphics, change the currently displayed video stream, etc. Applications are designed and specified with a domain-specific modelling language (that you develop) and delivered to digital televisions by inserting it as XML into the same broadcast stream that contains the video and audio content. Once running, an application can be updated by changing the XML in the broadcast stream. Applications can send reply messages to your system, usually via a standard modem built into the television. Sending these reply messages is very slow, and involves the viewer paying call charges. For these reasons, reply messages can only be used for viewer initiated actions, such as responding to a vote.

Code generation

You should generate code like the following as plain text, rather than using any special method for saving models as XML. This puts the various tools on an even footing, and demonstrates the code generation facilities better for all kinds of code and other output files. Handling white space and encodings is not vital, but the results should be machine and human-readable.

Listing 9.1: Sample Generated XML

```xml
<TVApp name=World Cup 2010>
  <Menu name=World Cup>
    <Vote name=Who will win?>
      <Choice name=Estonia/>
      <Choice name=Lithuania/>
    </Vote>
  </Menu>
  <Text name=World Cup trivia>Trinidad &amp;
```
9.1.2 Solution

The case study presented above is within the scope of Epsilon as it involves a number of different model management tasks such as model validation, model-to-model transformation, model merging and model-to-text transformation. Moreover, to implement some cases (e.g. the code generation case), more than one individual model management tasks need to be combined. This section demonstrates an implementation of the functional requirements (cases) discussed above using Epsilon languages combined using the workflow mechanism presented in Chapter 7. The aim of this implementation is to cover as many cases as possible in an integrated manner with minimal duplication of code.

Designing the TVApp DSL

The first step of the solution is to design a Domain Specific Language (DSL) that enables users to design Interactive TV Applications. The TVApp DSL has been specified atop EMF by defining its abstract syntax in terms of ECore (using Emfatic [151] as a convenience textual representation). A graphical overview of the DSL is presented in Figure 9.4.

Generating a TVApp model for a Sports Competition

To address Case 1, a new Competition metamodel has been designed for capturing information about competitions, groups and participating teams. A graphical overview of the metamodel is displayed in Figure 9.5. The design of the metamodel also satisfies the requirement of Case 2 for defining
Before the Competition model can be transformed into a TVApp model, it should be validated against a number of constraints so that potential errors are detected at this level and not propagated to the TVApp model. In Listing 9.2, an Epsilon Validation Language module is used for this purpose. The NameSpecified constraint in line 3 specifies that all named elements (that is Groups and Competitors) must specify a non-empty name. Moreover, in line 15, the NotEmpty constraint applies to Groups and specifies that each group must have at least one competitor. Here it is worth noticing that in its guard, the NotEmpty constraint requires that to apply to particular Group, the Group must first satisfy the NameSpecified constraint presented above. Finally, in line 29, the InUniqueGroup constraint checks that each competitor only participates in one Group.
Figure 9.5: The Competition Domain Specific Language

Listing 9.2: EVL module that validates a Competition model

```eclipse
context Competition!NamedElement {

  constraint NameSpecified {

    check : self.name.isDefined() and self.name.trim().length > 0

    message : self.eClass().name + ' must provide a name'

  }

}

class Competition!Group {

  constraintNotEmpty {

    guard : self.satisfies('NameSpecified')

    check : self.competitors.size() > 0

  }

}```
Having validated the input Competition mode, the Epsilon Transformation Language discussed in Section 6.2 is used to transform it into a TVApp model. The transformation displayed in Listing 9.3 defines that an Application and a Vote will be generated from each Competition, and that the Vote will contain a flattened sequence of Labels (one for each Group) and Choices (one for each Competitor in the group).

Listing 9.3: ETL transformation that transforms a Competition model into a TVApp model

```plaintext
rule Competition2Application
transform c : Competition!Competition
to a : TVApp!Application, v : TVApp!Vote {
a.name := c.name + ' Application';
v.name := 'Who will win the ' + c.name + '?';
a.contents.add(v);
for (g in c.groups) {
    message : 'Group ' + self.name + ' contains no competitors'
}
context Competition!Competition {
    constraint InUniqueGroup {
        guard : self.satisfies('NameSpecified')
        check : Competition!Group.allInstances.
            select(g|g.teams.includes(self)).size() <= 1
        message : 'Competitor ' + self.name +
            ' exists in more than one group.'
    }
}
```
v.contents.add(g.equivalent());
for (memb in g.members) {
    v.contents.add(memb.equivalent());
}
}
}

rule Competitor2Choice
transform co : Competition!Competitor
to ch : TVApp!Choice {
    ch.name := co.name;
}

rule Group2Label
transform g : Competition!Group
to l : TVApp!Label {
    l.name := 'Group ' + g.name;
}

The two steps are combined into a composite task that validates and - if all constraints are satisfied - then transforms the Competition model using the workflow presented in Listing 9.4. In lines 4 and 11, the involved models are loaded using the epsilon.loadModel task. In line 20 the Competition model is validated using the EVL module presented in Listing 9.2, and if all constraints are satisfied, in line 24, it is transformed into a TVApp model using the ETL module presented in Listing 9.3.

Listing 9.4: Workflow that integrates the validation and transformation steps
Integrating Live Reports

Case 3 requires that an external user must be able to update a particular text but not the structure of the application. To achieve this, the Report DSL presented in Figure 9.6 has been designed. The rationale is that particular users can be only granted the permission to compose Report models which will then be used to automatically update the original application model via merging. To satisfy the requirement set in Case 4, when merging the original TVApp with a Report, the original information contained in the Texts which the Report updates is not lost but instead is stored in the form of a new TextHistory model element. The merge has been specified using the ECL and EML languages modules displayed in Listings 9.6 and 9.5 respectively. An alternative to designing the one-metaclass Report DSL
would be to add the Report class to the existing TVApp DSL. However, it was deemed more appropriate with regard to modularity and separation of concerns to establish a new (albeit minimal) DSL instead.

As discussed in Section 6.5, EML operates in two steps. In the first step, correspondences are established between elements that need to be merged (using ECL or otherwise) and in the sequel corresponding elements are merged (using merge-rules) and non-matched elements are transformed into the target model (using transform-rules reused from ETL). In Listing 9.5 matching Reports from the Report model with Texts from an OldTVApp model is achieved through the MatchReportWithText ECL match-rule in line 1 that compares the names of the compared elements.

Then, in Listing 9.6 matching Reports and Texts are merged using the MergeReportWithText merge-rule in line 3 which specifies that the Text should be updated and that the old version of the text should be stored in the form of a new TextHistory model element with a proper information and revision number. The rest of the elements of the original TVApp model are copied to the target model using the transformation rules provided by the CopyTVApp module, displayed in Listing 9.7 which is imported by the EML module using the import statement in line 1.

Listing 9.5: ECL module that compares a TVApp with a Report model

```
rule MatchReportWithText
  match t : OldTVApp!Text
  with r : Report!Report {
    compare : r.name = t.name
  }
```

Listing 9.6: EML module that merges a TVApp with a Report model

```eureo
import 'CopyTVApp.etl';

rule MergeReportWithText
  merge ot : OldTVApp!Text
  with r : Report!Report
  into nt : NewTVApp!Text {
    nt.name := ot.name;
    nt.information := ot.information;
    nt.history ::= ot.history;

    var h : new NewTVApp!TextHistory;
    h.information := ot.information;
    h.revision := ot.history.collect(h|h.revision).max(0) + 1;
    nt.history.add(h);
    nt.information := r.information;
  }
```

Listing 9.7: ETL transformation that copies a TVApp module

```eureo
rule CopyApplication
  transform s : OldTVApp!Application
  to t : Target!Application {
    t.name := s.name;
    t.contents ::= s.contents;
  }

rule CopyText
  transform s : OldTVApp!Text
  to t : Target!Text {
    t.name := s.name;
    t.information := s.information;
    t.history ::= s.history;
  }

rule CopyTextHistory
  transform s : OldTVApp!TextHistory
  to t : Target!TextHistory {
    t.revision := s.revision;
  }
```
In case a report in the Report model does not match a text in the TVApp model, the report is (correctly) not added into the target TVApp model. To inform the user about such cases, the EVL module of Listing 9.8 is injected between the comparison and merging steps. The `RefersToValidText` constraint specified in line 3 examines the match-trace (trace) established by the comparison step and requires that each report must match with a Text from the TVApp model.

Listing 9.8: EVL module that validates a Report model against a TVApp
The ECL, EVL and EML modules presented above are integrated using the workflow of Listing 9.9. In lines 4, 11 and 19 the involved models are loaded. In line 30 the `Report` model is compared with the `OldTVApp` model using the ECL module of Listing 9.5.

Listing 9.9: Workflow integrating the comparison validation and merging steps

```
<?xml version="1.0"?>
<project default="main">
  
  <epsilon.loadModel name="Report" type="EMF">
    <parameter name="modelFile" file="models/Report.model"/>
    <parameter name="metamodelUri" value="ReportDsl"/>
    <parameter name="isMetamodelFileBased" value="false"/>
    <parameter name="readOnLoad" value="true"/>
  </epsilon.loadModel>

  <epsilon.loadModel name="OldTVApp" type="EMF">
    <parameter name="aliases" value="Source"/>
    <parameter name="modelFile" file="models/TVApp1.model"/>
    <parameter name="metamodelUri" value="TVAppDsl"/>
    <parameter name="isMetamodelFileBased" value="false"/>
    <parameter name="readOnLoad" value="true"/>
  </epsilon.loadModel>

  <epsilon.loadModel name="NewTVApp" type="EMF">
    <parameter name="aliases" value="Target"/>
    <parameter name="modelFile" file="models/TVApp2.model"/>
    <parameter name="metamodelUri" value="TVAppDsl"/>
  </epsilon.loadModel>

```
Generating XML from a TVApp model

To generate an XML document from a TVApp model, as required by the case study description, the intermediate XML metamodel displayed in Figure 9.7 is used. Thus, the TVApp model is transformed into an XML model (Listing 9.10) using the Epsilon Transformation Language, and then the textual representation of the XML model (Listing 9.11) is generated using the Epsilon Generation Language. This approach promotes modularity and
reuse by targeting XML-specific issues such as escaping and indentation in a separate transformation which can be reused as-is in other DSL to XML scenarios, and also complies with the requirement not to use a special (hard-coded) method for transforming XML models to text.
Listing 9.10: ETL transformation that transforms TVApp models to XML models

```plaintext
pre {
    var doc : new Xml!Document;
        allInstances.first().equivalent();
}

@abstract
rule NamedElement2Element
    transform ne : TVApp!NamedElement
    to n : Xml!Element {
        n.addAttribute('name', ne.name);
    }

rule Application2Element
    transform a : TVApp!Application
    to n : Xml!Element extends NamedElement2Element{
        n.name := 'Application';
    }
```

Figure 9.7: The XML Domain Specific Language
rule Vote2Element
  transform v : TVApp!Vote
to n : Xml!Element extends NamedElement2Element {

  n.name := 'Vote';
  n.contents := v.contents.equivalent();
}

rule Choice2Element
  transform c : TVApp!Choice
to n : Xml!Element extends NamedElement2Element {

  n.name := 'Choice';
}

rule Label2Element
  transform c : TVApp!Label
to n : Xml!Element extends NamedElement2Element {

  n.name := 'Label';
}

rule Text2Element
  transform t : TVApp!Text
to e : Xml!Element extends NamedElement2Element {

  e.name := 'Text';
  var text : new Xml!Text;
  text.cdata := t.information;
  e.contents.add(text);
}

rule Menu2Element
  transform m : TVApp!Menu
to e : Xml!Element extends NamedElement2Element {

  e.name := 'Menu';
}
operation Xml!Element addAttribute
  (name : String, value : String) {
    var attr : new Xml!Attribute;
    attr.name := name;
    attr.value := value;
    self.attributes.add(attr);
  }

The ETL transformation of Listing 9.10 demonstrates two important characteristics of ETL: rule inheritance and state-changing operations. By inheriting from the abstract rule NamedElement2Element, the rest of the rules are maintained simple and without duplication. Moreover, defining the addAttribute() state-changing operation simplifies the NamedElement2Element rule.

Listing 9.11: EGL template that generates XML text from XML models

```xml
<?xml version="1.0"?>
[%=Document.allInstances().first().rootElement.toString(0)%]

[%

operation Element toString(indent : Integer) : String {
  var str : String;
  str := indent.getIndent() + '<' + self.name.normalize();
  for (a in self.attributes) {
    str := str + ' ' + a.name.normalize() + '"' + a.value.normalize() + '"';
    if (hasMore){
      str := str + ' ';
    }
  }
  str := str + '>' +\r\n';
  for (c in self.contents) {
    str := str + c.toString(indent + 1);
  }
  str := str + indent.getIndent() + '</' +
    self.name.normalize() + '>\r\n';
  return str;
}]
```
Generating XML text from an XML model in Listing 9.11 involves much dynamic and little static text. Therefore, the vast majority of the serialization process is performed via string concatenation. By contrast, the header (processing instruction and comment) is only static text that is emitted as-is from the template.
Integrating the Model-to-Model and the Model-to-Text steps

To integrate the two steps presented above into a coherent process that transforms a TVApp model directly to textual XML, the Epsilon Workflow is used. Thus, in Listing 9.12, two tasks for loading the involved models (epsilon.loadModel), one that invokes the ETL transformation (epsilon.etl) and one that invokes the EGL model-to-text transformation on the intermediate XML model (epsilon.egl) are used.

Listing 9.12: The workflow that integrates the ETL and EGL tasks

```xml
<?xml version="1.0"?>
<project default="main">
  <epsilon.loadModel name="TVApp" type="EMF">
    <parameter name="modelFile" file="models/ChampionsLeague.model"/>
    <parameter name="metamodelUri" value="TVAppDsl"/>
    <parameter name="isMetamodelFileBased" value="false"/>
    <parameter name="readOnLoad" value="true"/>
  </epsilon.loadModel>

  <epsilon.loadModel name="Xml" type="EMF">
    <parameter name="modelFile" file="models/TVAppXml.model"/>
    <parameter name="metamodelUri" value="Xml"/>
    <parameter name="isMetamodelFileBased" value="false"/>
    <parameter name="readOnLoad" value="false"/>
  </epsilon.loadModel>

  <target name="main">
    <epsilon.etl src="TVApp2Xml.etl">
      <model ref="TVApp"/>
      <model ref="Xml"/>
    </epsilon.etl>

    <epsilon.egl src="Xml2Text.egl" target="output/TVApp.xml">
      <model ref="Xml"/>
    </epsilon.egl>
  </target>
</project>
```
Generating a Mock-up of the TV Application

A feature that is not required explicitly by the case study description, but which was regarded as particularly useful is to be able to preview a TVApp model using a mock-up that closely resembles the appearance of the final deployed application. Therefore, EGL has been used to compose a model-to-text transformation that generates a set of linked HTML screens that emulate the look-and-feel and functionality of the deployed application. Listing 9.13 presents the main template of the EGL solution that iterates the model and invokes the respective sub-templates (Text.egl, Menu.egl, Vote.egl) to generate the mockup HTML screens.

Listing 9.13: EGL template that generates mockup HTML screens for a TVApp model

```%
import 'include\Common.eol';

TemplateFactory.setTemplateRoot('workspace\MDD-TIF');
TemplateFactory.setOutputRoot('workspace\MDD-TIF\html');

var header : Template :=
    TemplateFactory.load('include\Header.egl');
var text : Template :=
    TemplateFactory.load('include\Text.egl');
var menu : Template :=
    TemplateFactory.load('include\Menu.egl');
var vote : Template :=
    TemplateFactory.load('include\Vote.egl');
var footer : Template :=
    TemplateFactory.load('include\Footer.egl');

var contents : String := 'default';

if (Application.isType(content) or Menu.isType(content)) {
    menu.populate('menu', content); 
    contents := menu.process();
}
```

-- Recursively generate the pages
-- for the contents of this
9.1.3 Summary

The case study presented in this section involves four different model management tasks, three of which comprise of more than one step, for each of which a different model management language is more appropriate. Table 9.1 provides an overview of the tasks and the languages involved in providing a solution for each one.

The solution has demonstrated that the layered architecture proposed in this work enables users to decompose complex tasks into steps that are easily manageable with one of the task-specific languages provided and then exploit the inherent interoperability of the individual languages to compose the steps.
### Table 9.1: Matrix of Tasks and applied Languages

<table>
<thead>
<tr>
<th>Task</th>
<th>Languages used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transform a Competition into a TVApp model</td>
<td>EVL, ETL</td>
</tr>
<tr>
<td>Provide Support for Live Reports</td>
<td>ECL, EVL, EML</td>
</tr>
<tr>
<td>Generate XML from a TVApp model</td>
<td>ETL, EOL</td>
</tr>
<tr>
<td>Generate Mock-up screens from a TVApp model</td>
<td>EGL</td>
</tr>
</tbody>
</table>

into coherent composite tasks using the proposed workflow solution.

## 9.2 Impact

This section evaluates the impact of this work on the Model Driven Engineering community.

### 9.2.1 The Eclipse GMT Research Incubator Project

In November 2006, Epsilon was invited to become a component of the Eclipse Generative Modeling Technologies (GMT) project \cite{152}. The GMT project is the research incubator project of the top-level Eclipse Modelling Project (EMP) \cite{153} which is generally recognized to be the most active community in the field of model driven engineering at present. As its research incubator, GMT hosts a limited number of research-oriented components (11 in May 2008) which have been selected through a rigorous process.

Since its incorporation in GMT, a growing community of users of Epsilon is forming. Evidence of this is the related activity in the respective public newsgroups (eclipse.modeling.gmt and eclipse.epsilon under news.eclipse.org) and the number of visits to the project website (around 14000 page loads, 7800 unique visitors and 2090 returning visitors from January 2007 to May 2008).\footnote{Metrics obtained using the StatCounter service (http://www.statcounter.com)}
9.2.2 Publications

The results of this research have been presented in a number of academic papers in journals, international conferences and workshops. At the time of writing (May 2008), the list of publications includes one book chapter, two journal papers, eight conference papers and nineteen workshop/symposium papers. In the following publications, the author of this thesis has been the primary contributor.


9th International Conference on Model Driven Engineering Languages and Systems (Models/UML 2006), Genova, Italy, October 2006.


In the following publications the author has been a secondary contributor. In each publication, a short description of the contribution of the author is provided.

In this work the author provided an early prototype which was then developed by the first author into the Epsilon Generation Language.


   In this work the author contributed to the establishment of a methodology for capturing rigorous model traceability information.


   In this work the author contributed to the definition of a methodology for building MDE traceability classifications.


   In this work the author contributed technical expertise in defining an EMF-based metamodel for software assurance and supporting model management operations.


   In this work, the Epsilon Transformation Language (ETL) was used to facilitate migration of models when the metamodels they conform to change. The contribution of the author was to propose model
transformation with ETL as a technique for achieving synchronization between models and metamodels and to implement a case study that demonstrated the practicality of the proposed approach.


   In this work the author contributed to the definition of a metamodel for dependability cases.


   In this work the author contributed experiences and expertise obtained through the development of the Epsilon Comparison and Merging languages in a common effort to establish a common vocabulary for model composition/integration.


   In this work the author demonstrated using EML for merging different versions of models.


   In this work the author contributed to demonstrating the usefulness of operational semantics for achieving traceability.
The number of publications suggests that the results of this research are well-communicated and established with the research community.

9.2.3 ModelWare and ModelPlex

Languages from Epsilon have been used in the context of the EU ModelWare [40] and ModelPlex [39] projects to implement case studies established by industrial partners such as IBM, SAP, Telefonica, Western Geco (WGO) and Thales Information Systems. Most notably, EOL has been used to perform performance simulation on UML2 Activity Diagrams in the context of a SAP-provided case study, EVL, ETL and EOL are being used in the context of a sensor network modelling case study provided by Western Geco, and ECL and EML have been used in a protocol composition case study in collaboration with SINTEF. Moreover, concepts and facilities from EVL are being considered by ModelPlex partners for proposal to the OMG as extensions for future revisions of the OCL standard.

This section provides an overview of one of the cases where Epsilon was used by the author in ModelPlex: the SAP UML2 Activity Diagram performance simulation case study.

**SAP: UML2 Activity Diagram Simulation**

The aim of this case study was to simulate UML2 activity diagrams annotated with performance-related information, in order to identify potential performance problems during early design phases. For this purpose, the SAP team involved in ModelPlex have designed a UML2 profile that allows users to annotate activity control flows with probabilities to simulate non-deterministic execution, and annotating activities with three types of costs: the CPU time it requires to execute, the time it requires to access the underlying database and the number of database table rows it needs to access. An exemplar UML2 activity diagram provided for this case study is displayed in Figure 9.8.

---

2The performance-related information is included in the underlying model but does not appear in the graphical notation.
Figure 9.8: Exemplar Performance-Annotated UML2 Activity Diagram
The implementation of this case study using Epsilon consisted of two steps. In the first step, EVL was used to validate the activity diagram to ensure that the sum of probabilities of the non-deterministic transitions of each decision node is exactly 1. Listing 9.14 demonstrates the Probabilities-SumTo1 constraint specification using EVL.

Listing 9.14: Performance-Annnotated Activity Diagram Constraints using EVL

```eclipse
import 'UML2.lib.eol';

class DecisionNode {

    constraint ProbabilitiesSumTo1 {

        guard : self.outgoing.exists(cf|cf.
            stereotypeApplication('ExecutionProbability').
            isDefined())

        check {
            var sum : Real := 0;
            for (cf in self.outgoing) {
                var sa := cf.getStereotypeApplication('ExecutionProbability');
                if (sa.isDefined()) {
                    sum := sum + sa.Probability;
                }
            }
            return sum = 1.0;
        }

        message : 'Sum of probabilities of decision node ' + self.name + ' is ' + sum
    }
}
```

The next step was to simulate the execution of the model. For this purpose, non-invasive user defined operations (discussed in Section 5.2.3)
were defined on the executable parts of the model using EOL. Listing 9.15 demonstrates the simulation specification in EOL. In line 17 the \texttt{DecisionNode.execute()} operation randomly selects one of the outgoing control flows of the decision node (based on their probabilities) and calls the \texttt{walk()} operation on it. In line 62 the \texttt{walk()} operation executes the target of the control flow. Operation \texttt{Action.execute()} in line 29 executes an action by adding its costs to the global variables defined in lines 5–7. Finally, the \texttt{JoinNode.execute()} operation defines that all incoming control flows must be first traversed before the outgoing control flows of a join-node can be executed.

Listing 9.15: Simulation using EOL

```java
import 'UML2.lib.eol';

var initialNode : InitialNode := InitialNode.allInstances().first();

var cpuTime : Integer := 0;
var dbAccess : Integer := 0;
var dbRows : Integer := 0;
var joinNodes : new Native('java.util.HashMap');
var finished : Boolean := false;

initialNode.execute();
'Simulation trace : '.println();
(' CPUTime : ' + cpuTime).println();
(' DBAccess : ' + dbAccess).println();
(' DBRows : ' + dbRows).println();

operation DecisionNode execute() {
    var poll : Sequence;
    for (cf in self.outgoing) {
        for (i in Sequence{0..(cf.getStereotypeApplication('ExecutionProbability').Probability * 100).asInteger() - 1}){
            poll.add(cf);
        }
    }
    poll.random().walk();
}```
operation Action execute() {
    self.name.println();
    cpuTime := cpuTime + self.getCPUTime();
    dbAccess := dbAccess + self.getDBAccess();
    dbRows := dbRows + self.getDBRows();
    self.outgoing.first().walk();
}

operation ControlNode execute() {
    for (o in self.outgoing) {
        o.walk();
    }
}

operation JoinNode execute(cf : ControlFlow) {
    if (joinNodes.get(self).isUndefined()) {
        joinNodes.put(self, Set{cf});
    } else {
        joinNodes.get(self).add(cf);
    }
    if (joinNodes.get(self).size() = self.incoming.size()) {
        for (o in self.outgoing) {
            o.walk();
        }
        joinNodes.put(self, Set{ });
    }
}

operation ActivityFinalNode execute() {
    finished := true;
}

operation ControlFlow walk() {
    if (finished) {
        return;
    -- If a final node has been reached
    -- by some other path return
    if (finished) { return; }
if (self.target.isTypeOf(JoinNode)) {
    self.target.execute(self);
}
else {
    self.target.execute();
}

A sample output of the simulation program appears in Listing 9.16.

Listing 9.16: Sample output of the simulation program of Listing 9.15

CreateOrder
Check Data Completness
ChangeRequest
Change Order
Check Data Completness
ChangeRequest
Change Order
Check Data Completness
Check Consistency
Check Product Availability and Reserve
ConfirmOrder
Request Invoicing
Request Requirement, Reserve Fulfill
Simulation trace :
CPU Time : 2440
DB Access : 357
DB Rows : 347

It is worth noting that despite their different purposes, both the EVL constraints module in Listing 9.14 and the EOL simulation module in Listing 9.15 import the Uml.lib.eol library which is displayed in Listing 9.17 to reuse the `getStereotypeApplication()` operation it contains.

Listing 9.17: The shared Uml.lib.eol library

```java
operation Element getStereotypeApplication(stereotype : String) {
    var s : Stereotype := self.getAppliedStereotypes().
    select(s|s.name = stereotype).first();
    if (s.isDefined()) {
```
return self.getStereotypeApplication(s);
}
}

Through this case study, the capabilities of Epsilon for programmatic model manipulation were demonstrated to the SAP team which provided positive feedback and expressed interest for further experiments in the context of performance-driven model engineering using Epsilon.

9.2.4 External References

Through a web survey, a number of external publications (currently [154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178]) in which the respective work is compared to, or uses, the work performed in the context of this thesis have been identified. This provides further evidence of the novelty of this work and its impact and acceptance in the MDE research community.

9.2.5 Uses in other Projects

INESS Project

The department of Computer Science will explore the use of Epsilon in the context of the European FP7 project “INESS” (Integrated European Signalling System) (starting in autumn 2008) for validation of functional requirements for rail signalling systems across Europe. This will include: capture of functional and safety requirements for rail signalling systems; simulation of these requirements; transformation of models into formal representations (e.g., in B and Circus); and generation of reports. As such, the project expects to use Epsilon languages like ETL, EGL, and EVL.

SSEI Project

In the context of the SSEI project\(^3\), the department of Computer Science will use Epsilon for model-based systems integration, involving systems that

\(^3\)http://www.ssei.org.uk
exist solely as code (both source and binary) as well as models stored in a variety of formats. The project envisions using ECL for model comparison of both structure and behaviour; and EML for integration. Additional effort will need to be made to support the production of models from text representations; this may involve integration of components from the AMMA and openArchitectureWare platforms, or the development of new Epsilon components.

9.2.6 Internal and External Collaborations

The results of this work have been used in a number of internal and external collaborations. This section summarizes the most important of them.

University of Seville, Group MaCMAS (Methodology for Analysing Complex MultiAgent Systems) The MaCMAS group has implemented an extension to the ArgoUML modelling tool to support complex multi-agent systems. During this collaboration, their modelling tool was enhanced with support for specifying and executing Epsilon Object Language (EOL) programs which allowed the group to further automate the model refinement process.

Boris Gruschko, SAP, Karlsruhe In this work ETL was used to perform automated migration of models as a response to changes in the metamodels they conform to. The outcome of this work was presented in [26].

Philippa Conmy, HISE, York In this work EOL was used to perform mutation of processor-network models for failure analysis and ETL was used to transform models into a suitable format for a proprietary tool that implements FPTC (Failure Propagation Transformation Calculus).

George Despotou, HISE, York In this work, EVL constraints were used to express the constraints of a metamodel for Dependability Cases and EWL wizards were used to automate the model composition and refinement process.
9.3 Evaluating Reuse and Interoperability

The research hypothesis stated that despite their differences and task-specific requirements, ...all those different model management tasks can be supported with a family of integrated task-specific languages that builds on a platform that provides a set of common reusable features to maximize reuse, uniformity and interoperability. This section evaluates reuse and interoperability in the proposed platform both from a language engineering and a user perspective and demonstrates how the architecture and organization of Epsilon contributes positively to both of those essential quality attributes.

9.3.1 Language Engineering Perspective

To justify the claim for reuse from a language engineering perspective, the amount of infrastructure functionality that can be reused when implementing task-specific languages should be considerable. This section provides evidence about reuse metrics in the Epsilon prototype. To measure reuse, this section provides metrics of the amount of source code that has been required to provide tool-support for the base language (EOL) and the task-specific languages built atop it.

Runtime

Table 9.2 shows the total lines of code required to implement the runtime (interpreter and internal object model) of each language in the platform. These figures suggest the benefit from reuse as the size of the runtime of each task-specific language is around 50% of the size of the shared infrastructure.

However, not all source code required for constructing a language runtime has been hand-crafted. Instead, a part of the code has been generated
Table 9.2: Lines of Java Code for each Language Runtime

<table>
<thead>
<tr>
<th>Language</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>EOL</td>
<td>13335</td>
</tr>
<tr>
<td>ETL</td>
<td>6683</td>
</tr>
<tr>
<td>EML</td>
<td>6878</td>
</tr>
<tr>
<td>EVL</td>
<td>7216</td>
</tr>
<tr>
<td>ECL</td>
<td>6804</td>
</tr>
<tr>
<td>EWL</td>
<td>6169</td>
</tr>
<tr>
<td>EGL</td>
<td>2609</td>
</tr>
</tbody>
</table>

Table 9.3: Lines of Grammar and Generated Code for each Language Runtime

<table>
<thead>
<tr>
<th>Language</th>
<th>Grammar LOC</th>
<th>Generated LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>EOL</td>
<td>1005</td>
<td>5441</td>
</tr>
<tr>
<td>ETL</td>
<td>95</td>
<td>5962</td>
</tr>
<tr>
<td>EML</td>
<td>64</td>
<td>6141</td>
</tr>
<tr>
<td>EVL</td>
<td>119</td>
<td>6400</td>
</tr>
<tr>
<td>ECL</td>
<td>109</td>
<td>6055</td>
</tr>
<tr>
<td>EWL</td>
<td>74</td>
<td>5904</td>
</tr>
<tr>
<td>EGL</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

from the grammar that specifies the textual concrete syntax of each language. Table 9.3 demonstrates the lines of code of the grammar of each language and the code generated from it by the parser generator component (which is ANTLR [146] for the specific implementation).

To obtain more representative source code metrics, the size of the generated code for each language parser is replaced with the size of the grammar from which it has been automatically generated. The updated results appear in Table 9.4.

According to the results, to construct a new task-specific language, an approximate figure of 800 lines of hand-crafted Java code and ANTLR grammar is required. This figure excludes the Epsilon Generation Language (EGL) which deviates significantly by requiring almost three times as much code (2609 lines) to implement. The reason for this is that unlike the rest of the languages which are rule-based, EGL is of a radically different nature.
and as such it requires a considerable amount of code to accommodate the requirements of the task it has been designed for (file management, content preservation etc.).

**Development Tools (DT)**

Table 9.5 provides metrics for the lines of code required for the Eclipse-based development tools (DT) of each language.

According to the figures above, the layered architecture has an impact on the amount of code required to implement development tools for task-specific languages.

Summing up the figures provided in Tables 9.4 and 9.5 produces the final figures of lines of code required to implement both runtime and end-user tool-support a new task-specific language are displayed in Table 9.6.

The metrics displayed in Table 9.6 show that, with the exception of
Table 9.6: Total Lines of Code Required for Supporting each Language

<table>
<thead>
<tr>
<th>Language</th>
<th>Runtime LOC</th>
<th>DT LOC</th>
<th>Total LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>EOL</td>
<td>8899</td>
<td>2990</td>
<td>11889</td>
</tr>
<tr>
<td>ETL</td>
<td>816</td>
<td>439</td>
<td>1255</td>
</tr>
<tr>
<td>EML</td>
<td>801</td>
<td>325</td>
<td>1126</td>
</tr>
<tr>
<td>EVL</td>
<td>935</td>
<td>457</td>
<td>1392</td>
</tr>
<tr>
<td>ECL</td>
<td>858</td>
<td>374</td>
<td>1232</td>
</tr>
<tr>
<td>EWL</td>
<td>339</td>
<td>552</td>
<td>891</td>
</tr>
<tr>
<td>EGL</td>
<td>2609</td>
<td>921</td>
<td>3530</td>
</tr>
</tbody>
</table>

EGL as discussed above, implementing a new task-specific language requires 1000-1500 lines of grammar and application code which is roughly 10% of the infrastructure code. This validates the original hypothesis that different model management tasks share a significant amount of functionality and can therefore be implemented with enhanced reuse atop a platform, such as Epsilon, that provides a common shared infrastructure.

9.3.2 End-User Perspective

In Section 2.5 a number of reuse, consistency and integration problems were identified from a user perspective for the current state-of-the-art model management languages. This section summarizes the identified challenges and demonstrates how the architecture of Epsilon addresses them from a user perspective.

Syntax Consistency

As demonstrated in Listings 2.6, 2.7 and 2.8 languages that target different tasks often support similar but inconsistent concrete syntaxes. Therefore, a user cannot reuse a piece of functionality (e.g. an operation/method) originally defined in the context of a model to text transformation program, in a model to model transformation or model validation program. As a result, users need to maintain copies of the same functionality in many different, similar - but inconsistent - languages.

Epsilon addresses this issue effectively since all languages of the platform
inherit their abstract and concrete syntax as well as their operational semantics and implementation from the core EOL language, and as such they are inherently consistent with each other. As an example, in Listing 9.18, the hasStereotype operation is defined for UML model elements in the UML-Operations.eol file. Then, in the model to model transformation of Listing 9.19 UMLOperations.eol is imported and the hasStereotype operation is invoked in line [3]. Similarly, the same operation is reused in the context of a model validation module in line [6] of Listing 9.20 and in a model to text transformation in line [5] of Listing 9.21.

Listing 9.18: Definition of the hasStereotype() operation in UMLOperations.eol

```eol
operation UML!ModelElement hasStereotype(s : String) : Boolean {
  return self.getAppliedStereotypes()
    .exists(st|st.name = s);
}
```

Listing 9.19: Excerpt of a UML to DB transformation in ETL

```eol
import 'UMLOperations.eol';

rule Class2Table
  transform c : UML!Class
to t : DB!Table
  extends ModelElement2NamedElement {
    guard : c.hasStereotype('table')
    t.database ::= c.namespace;
    var idCol := new DB!Column;
    idCol.name := 'id';
    t.columns.add(idCol);
  }
```

275
Listing 9.20: Excerpt of a UML validation module in EVL

```java
import 'UMLOperations.eol';

context UML!Class {
    constraint NoAbstractTables {
        check : self.hasStereotype('table')
            implies (not self.isAbstract)
        
        message : 'Class ' + self.name +
            ' is abstract and also' +
            ' stereotyped as <<table>>'
    }
}
```

Listing 9.21: Excerpt of a UML to Java model to text transformation in EGL

```java
[%import 'UMLOperations.eol';%]

public class [%=c.name%] {
    ...
    [%if (c.hasStereotype('table')) {%
    public void persist() {
        ...
    }
    [%}]
    ...
}
```

**Runtime Interoperability**

Another issue discussed in Section 2.5 is that of runtime interoperability between different model management languages that enables to compose individual model management programs into complex workflows. An additional consideration is that since model management tasks actually make up only for a small fraction of the tasks in a software development process, model management languages should integrate with non-MDE tasks such as code compilation, version control management etc.
There are three approaches to runtime interoperability: file-based, adapter-base and direct interoperability. In file-based interoperability, each model management language is responsible for loading the models it needs to operate on, performing operations on them and then storing them back so that the next model management language can proceed, in a way similar to the operation of UNIX pipes. The main downside of this approach is that it requires models to be loaded and stored from the filesystem to the memory and vice-versa multiple times which, is costly both in terms of memory footprint and performance. In adapter-based interoperability, different model management languages use similar - but not identical - model representation formats and as such adapters can be used perform in-memory transformations that enable a language to reuse a model loaded by another language in a previous step. As an example, ATL and MOFScript operate on EMF-based models, but each one provides a different wrapper around the native EMF objects that represent model elements in memory. Therefore, it is feasible to construct an adapter that produces an ATL-wrapped EMF model from a MOFScript-wrapped one and vice versa. This approach overcomes the issue of needing to load and store models multiple times within a workflow. On the other hand, the construction of such an adapter is a non-trivial process and also the adapter needs to be maintained as the APIs of the respective languages evolve.

Epsilon addresses this issue by providing direct interoperability since all languages are built on top of a common model connectivity framework, and as such they share the same internal representation scheme for models and a common implementation. As such, when a model is loaded in memory, it can be accessed and modified by any language of the platform - thus making loading and storing models multiple times in the context of the same workflow unnecessary. Also, by sharing implementation modules expressed in different languages of the platform can interoperate directly with each other at runtime by sharing variables as discussed in Section 7.4.
9.4 Comparison with other Platforms of Model Management Languages

This section provides an in-depth comparison of Epsilon with a number of existing platforms of model management languages identified in Section 2.5.4 and demonstrates how Epsilon relates to them in terms of scope and capabilities.

9.4.1 The openArchitectureWare (oAW) platform

The oAW platform provides languages for model-to-text generation (XPand) and intra-model validation (Check) that build on a common OCL-based expression language (XTend). XTend can also be used as a general-purpose imperative language for specifying purely imperative model to model transformations. oAW also provides a language for text-to-model transformation (XText) which interoperates with the core expression language of the platform to enable users specify reference resolution functionality during the construction of a model from an abstract syntax tree using XTend. Finally, it provides a custom workflow language for constructing and executing workflows that involve different model management tasks.

Compared to oAW, Epsilon provides a wider range of task-specific languages that includes languages for tasks such as model comparison and merging, user driven in-place model transformation and hybrid model transformation. Moreover, EVL provides better support for model validation than the respective Check language in oAW as it provides additional features such as constraint dependency management, and support for semi-automatically repairing failed constraints, as discussed in Section 6.3. Finally, compared to XTend, EOL provides additional novel features such as support for transactions, extended properties and context-independent user input.

On the other hand, as discussed in Section 3.3, Epsilon operates in the models technical space and thus it does not provide a language for text to model transformation similar to XText. Moreover, oAW languages are strongly-typed and this has facilitated the development of elaborate devel-
oper tools that provide features such as code completion and type checking.

9.4.2 The AMMA platform

The AMMA platform consists of the ATL model to model transformation language, the KM3 metamodeling language, the TCS language that supports bidirectional model to text and text to model transformations, and the AMW tool that supports model weaving. Apart from its intended use as a model transformation language, ATL has also been used as a validation language, by transforming a number of source models into a target model that contains errors and warnings as model elements, and as a model to text transformation language.

With regard to providing support for additional model management tasks, such as model comparison and merging, AMMA follows a different approach than Epsilon. While in Epsilon each task is assigned a new task-specific language, in AMMA the concept of Higher Order Transformations is used. Under this approach, the requirements of the task are captured in the form of a weaving model which is then transformed using ATL into an ATL transformation that eventually performs the task.

The main advantage of this approach is that only one model management language needs to be maintained. However, to implement this approach in practice, users need to have knowledge of the (non-trivial) metamodel of ATL as this is essential for writing Higher Order Transformations (HOTs) that generate ATL transformations as output. Another concern is that this approach, typically produces ATL transformations that are particularly complex and difficult to understand and debug. Moreover, as there are no stable general purpose higher-order transformations for tasks such as model comparison and merging, users need to implement such transformations themselves. By contrast, Epsilon provides dedicated and ready to use languages for these tasks, as well as support for implementing languages for additional tasks with reduced effort due to the extensive reusable infrastructure it provides.
9.4.3 The OMG platform of languages

The Object Management Group has proposed three languages for supporting the tasks of model validation (OCL) [57], model-to-model transformation (QVT) [82] and model-to-text transformation (MOF2Text) [180]. QVT is further split into the QVTO (operational) and QVTM (relational) languages. QVTO is an imperative model transformation language built on top of an imperative extension of OCL (IOCL), while QVTR is a declarative model mapping language that can be executed in two modes: enforcing and checking. In the enforcing mode, a QVTR specification acts as a model transformation while in the checking mode it acts as a comparison as it attempts to discover instantiations of the relations it specifies in the models against which it is checked. As discussed in [82], the checking mode of QVTR is available only for models previously transformed using the QVTR enforcing mode as it requires access to the established transformation trace, unlike ECL which can compare any pair of models, regardless of the way in which they have been constructed.

Similarly to oAW and AMMA, the OMG family of languages does not encompass dedicated languages for the tasks of comparison, merging and user-driven in-place model transformation. Moreover, as discussed earlier, the validation language of Epsilon provides a number of additional features to the OMG OCL.

A major issue with the OMG platform of languages is that until recently it has only existed in the form of specifications (some of them demonstrating severe flaws as demonstrated in [181]), and still some of them are still at a very early stage of implementation. For example, the implementation of the MOF2Text model-to-text transformation language implemented by the Eclipse M2T project has not been yet released. Also, at the time of writing this report there is no complete implementation that supports both the operational and relational sub-languages of QVT. There has recently been an effort to intensify collaboration between the OMG and the Eclipse foundation so that OMG standards are implemented by Eclipse and such flaws are identified and corrected in the specifications early in the process.
9.4.4 Comparison Summary

This section composes the results of comparing Epsilon with other model management platforms. Table 9.7 provides a matrix of the model management tasks that each platform natively supports. Here it should be stressed again that in principle all tasks can be implemented with all platforms since all platforms contain a general purpose imperative model management language (EOL for Epsilon, XTend for oAW, the imperative subset of ATL for AMMA and IOCL for the OMG platform of languages). Therefore, Table 9.7 only refers to the tasks that are supported using a targeted task-specific language and clearly demonstrates that Epsilon provides support for a wider range of consistent and interoperable model management languages than all the other compared platforms.

<table>
<thead>
<tr>
<th>Platform</th>
<th>MV</th>
<th>M2M</th>
<th>M2T</th>
<th>T2M</th>
<th>MM</th>
<th>MC</th>
<th>IPMT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Epsilon</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>oAW</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMMA</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OMG</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 9.7: Supported Tasks Matrix

9.5 Limitations and Shortcomings

This section presents the main limitations and shortcomings of the proposed approach and reference implementation, explains why these limitations could not be addressed in the context of this work, and provides directions for overcoming them in future design and implementation iterations.

---

4MV: Model validation, M2M: Model to model transformation, M2T: Model to text transformation, T2M: Text to model transformation, MM: Model merging, MC: Model comparison, IPMT: User driven in-place model transformation
9.5.1 Lack of Formal Execution Semantics

As discussed in Section 5.2, different approaches can be used to specify the execution semantics of software languages. With respect to the time constraints of this project, a decision had to be made early in the process as to whether formal semantics (e.g. translational/denotational semantics) or a reference implementation should be employed to specify the execution semantics of the languages designed within the project.

The decision to provide a reference implementation instead of a formal semantics was predominantly influenced by the setting in which the project was carried out. More specifically, the project was realized within two large EU projects (ModelWare and ModelPlex) where research results needed to be exploitable by industrial partners on their case studies. A concrete reference implementation made it possible to implement several case studies provided by industrial partners and thus, receive valuable feedback from them that has partly driven the evolution of the architecture of the platform and the individual languages to their current state. Also, through the reference implementation, Epsilon has become visible and accessible to external users through the respective Eclipse GMT component, as discussed in Section 9.2.1. This has generated substantial feedback from external users, that has been valuable for identifying and repairing design and technical defects in the languages.

At this point it should be stressed that the aim of this discussion is to justify the choice of a reference implementation over a formal semantics under the imposed constraints, not to underrate the value of formal semantics. An established formal semantics of the languages would enable formal analysis, refinement and proof of model management programs, which are all desirable capabilities in the context of model driven engineering, particularly when used to construct safety- (and otherwise) critical systems.
9.6 Reference Implementation

This aim of this section is to evaluate the reference implementation that has been realized to examine the validity of the hypothesis.

As examples of hierarchically layered families of languages, particularly on model management, are practically non-existent in the literature, it was realized early in the process that defining the final syntax and semantics of the languages would require many change cycles - as it eventually has.

To reduce ripple effects triggered by frequent changes in the languages, the architecture and technical design of the platform were predominantly driven by modifiability and extensibility. Therefore, the reference implementation intentionally fails to provide support for lower priority - but nevertheless useful - features such as static type checking, context-sensitive editing features and debugging support.

Extensive ripple effects introduced by such facilities would influence the evolution of the languages negatively since they would require significant effort to address and thus gradually become a burden for experimenting with novel features that would require changes to the syntax and the semantics of the languages. This intuition proved to be reasonable during the development life-cycle of the languages and their supporting tools. Due to its flexibility, the established architecture enabled experimentation with novel features such as user-interactivity, transactions and native-object access without significant ripple effects.

Nevertheless, the flexible architecture of the implementation has had an impact on performance and usability. By choosing a lightweight interpreter-based approach instead of a compiler-based approach, the runtime performance of the languages is impacted negatively, as expected. Moreover, the development tools lack productivity features such as automated support for refactoring and coding assistance which are important for large-scale industrial use.

Overall, the trade-off between flexibility and performance/usability is considered to be fair as the purpose of the project was not to produce industrial strength tools, but to validate the hypothesis that a layered platform
of languages with emphasis on reuse, uniformity and consistency can be used to address diverse model management tasks in a meaningful way.

9.7 Chapter Summary

This chapter has confirmed the validity of the hypothesis stated in Chapter 3 using a number of criteria such as: applicability in a complex externally-defined case study, reuse metrics, publications and acceptance by the research community, establishment of a user community, internal and external collaborations, and use of the reference implementation for several case studies in the context of the ModelWare and ModelPlex research projects.

The evaluation has demonstrated significant benefits in terms of applicability, reuse, modularity and interoperability and has identified limitations such as the lack of a formal semantics and the need for more sophisticated development tools.
Chapter 10

Conclusions and Further Work

This thesis addressed the following issues:

- A Model Driven Engineering process typically involves a number of different model-related tasks such as model-to-model transformation, model comparison, model merging, model-to-text transformation, model validation and in-place model transformation.

- Many different modelling languages/technologies may be used to capture models of interest in the context of the same Model Driven Engineering process.

- Each model management task demonstrates specific characteristics and requirements which are typically best addressed by task-specific languages (as opposed to a general-purpose language that fits all the tasks).

- To achieve usability and reuse, the different task-specific languages that are used in the context of the same process should be consistent with each other.

- To enable integration and to manage complexity, it is essential that individual tasks can be seamlessly combined into complex workflows.
These issues were studied with respect to the following hypothesis, originally stated in Section 3.2:

*Despite their individual requirements and characteristics, a wide range of task-specific modelling languages share a significant number of common features, and therefore, instead of developing each language separately, it is beneficial in terms of reuse, uniformity and interoperability to develop them atop a platform that provides a reusable set of commonly required features.*

The objectives of the thesis were:

1. To develop a platform atop which uniform, interoperable and reusable languages can be developed.

2. To use the platform to develop task-specific languages that address the, now largely unsupported, tasks of inter-model consistency checking, model comparison, model merging and in-place model transformation.

3. To use the platform to develop uniform task-specific languages for tasks that are already supported by existing languages (e.g. model-to-model transformation, model validation).

4. To develop an orchestration and coordination framework that enables composition of individual model management tasks implemented using languages of the platform into coherent workflows.

This chapter summarizes the conclusions and findings of the research that was performed in order to assess the validity of this hypothesis. The rest of the chapter is organized as follows. Section 10.1 presents the findings of the background and literature review. Section 10.2 summarizes the contributions of the proposed solution and the findings obtained during the life-cycle of the prototype implementation. Finally, Section 10.4 provides directions to further work both in the scope of the proposed solution and in a broader context.
10.1 Review Findings

In Chapter 2 a review of the background and existing work in the field of model management was performed. During the review, it was identified that a Model Driven Engineering process can involve a number of distinct model management tasks. A non-exhaustive list of the most commonly performed tasks follows:

- Model-to-model Transformation
- In-place model Transformation
- Model Comparison
- Model Merging
- Model-to-text Transformation
- Model Validation

The next step involved a review of existing mechanisms, tools and languages, that have been proposed for automating individual model management tasks. Through the review it was observed that each task demonstrates unique characteristics and requirements for which task-specific languages are better suited than general-purpose languages.

In the sequel, a survey of existing task-specific model management languages was performed. Through this survey, a number of languages that support tasks such as model validation, model-to-model transformation and model-to-text transformation were identified. By contrast, no task-specific languages were identified for tasks such as interactive in-place model transformation, model comparison and merging.

Another finding of the survey was that existing task-specific model management languages have been predominantly developed in isolation. Three particular shortcomings of this trend were observed:

- Significant duplication exist between different languages.
Most typically re-implement a subset of OCL for model navigation and querying.

Each language re-implements technology-specific model-access features.

- Languages for different tasks are inconsistent with each other and therefore prohibit reuse.

  The subset of OCL they use is not consistent across different languages.

  Commonly needed constructs, which are however not provided by OCL, (e.g. model modification constructs) are diversely implemented in different languages.

- The inherent inconsistency also renders combining individual tasks implemented using different model management languages into complex workflows particularly challenging.

The yet unexploited potentials for reuse and uniformity identified during the field review were summarized in Chapter 3 and the research hypothesis was stated.

10.2 Proposed Solution and Prototype Findings

In Chapter 4 a high-level architectural plan of a platform of integrated task-specific languages for model management that share abstract and concrete syntax as well as operational semantics and implementation was outlined. The proposed platform contains five distinct layers which were further elaborated in Chapters 5, 6, 7 and 8. The following sections summarize the intent of each layer and its contribution to the overall solution.

10.2.1 Model Connectivity Layer

Regardless of their task-specific characteristics, all model management languages need to access the contents of the models they operate on. To reduce
coupling between specific model management technologies and model man-
agement languages, an abstraction layer (Model Connectivity Layer) layer
was proposed in Section 4.4 and elaborated in Section 5.1. The aim of this
layer was to offer uniform model access services such as the retrieving all
instances of a given type, checking for ownership, retrieving and setting the
values of model element properties, and instantiating and managing trans-
actions.

10.2.2 Common Navigation and Modification Language

Similarly, despite their differences and task-specific requirements, all model
management languages include a mechanism for navigating, querying and
(in most cases) modifying models. In existing languages, a custom subset of
OCL has been traditionally re-implemented for this purpose, thus leading
to the interoperability and uniformity issues discussed above. To overcome
this issue, a feature-rich reusable OCL-based core language (the Epsilon
Object Language) that offers features such as first-order logic expressions,
model querying, navigation and modification, user-interaction was proposed
in Section 4.5 and further elaborated in Section 5.2.

10.2.3 Task-Specific Languages

A number of task-specific languages built on the infrastructure discussed
above were also proposed in Section 4.6 and were individually discussed in
detail in Sections 6.1 - 6.5. By building on this common infrastructure, all
task-specific languages inherit a number of features such as the ability to
manage an arbitrary number of models of diverse metamodels and modelling
technologies simultaneously, to interact with the user at runtime, to use
native objects for delegating computationally expensive and otherwise out-
of-scope tasks to the underlying platform. The following paragraphs provide
a short summary of each task-specific language.
Epsilon Comparison Language (ECL)

ECL, discussed in detail in Section 6.1, is a rule-based hybrid language for establishing correspondences between elements of an arbitrary number of input models. Similarly to ETL, ECL features rule and module-level inheritances as well as advanced concepts such as lazy, abstract and greedy comparison rules.

Epsilon Validation Language (EVL)

EVL, discussed in detail in Section 6.3, is a language that targets the task of model validation. EVL provides a number of novel features such as inter-model inconsistency detection, inconsistency repairing facilities, constraint dependency management and separation between critical and non-critical constraints.

Epsilon Transformation Language (ETL)

ETL, discussed in detail in Section 6.2, is a rule-based hybrid model transformation language that can be used to transform an arbitrary number of source models into a number of target models. ETL features rule and module-level inheritance as well as advanced concepts such as lazy, abstract and greedy transformation rules.

Epsilon Wizard Language (EWL)

EWL, discussed in detail in Section 6.4, is a hybrid language for performing user-driven in-place model transformations such as model element creation or refactoring.

Epsilon Merging Language (EML)

EML, discussed in detail in Section 6.5, is a rule-based hybrid language for merging an arbitrary number of models based on identified correspondences (calculated using ECL or otherwise).
Three task-specific languages have been proposed in this work for tasks that were previously largely unsupported (model comparison, merging and in-place transformation). Also, a model validation language (EVL) that improves the level of support for model validation in comparison to OCL by providing features such as inter-model consistency checking and automated inconsistency resolution mechanisms has been proposed. Finally, a model-to-model transformation language (ETL) that is of similar capabilities to the state-of-the-art model transformation languages has been proposed.

Apart from the benefits of uniformity, consistency and interoperability, the layered architecture of Epsilon demonstrated another significant benefit. More specifically, due to the layered nature of the platform, features added to the core language (EOL) were immediately available from within the context of task-specific languages. For example, when user interaction features were added to EOL, all task-specific model management languages were instantly converted into interactive as it has been shown in Sections 6.1.5 and 6.2.6. A similar case was when support for accessing the underlying platform using native objects was added to EOL. This immediately made it possible to implement more sophisticated string comparison algorithms in ECL (as discussed in Section 6.1.4).

10.2.4 Orchestration Workflow Layer

To coordinate the execution of model management tasks implemented using different model management languages, a workflow mechanism was proposed in Section 4.7 and elaborated in Chapter 7. The workflow provides facilities that enable seamless integration of individual model management tasks implemented using task-specific model management languages. The most important features of the workflow include a common model repository that is accessible by all the tasks participating in a workflow, a variable sharing mechanism that enables runtime communication between different tasks and the inherent ability - by building atop ANT - to integrate seamlessly with
10.2.5 Development Tools Layer

To enable users to compose and execute Epsilon programs, Eclipse-based development tools have been constructed as part of the reference implementation of the platform. The Epsilon development tools, discussed in Section 8.3 include editors, outline viewers, launch configurations and delegates and an interactive console.

10.3 Evaluation Results

In Chapter 9, the validity of the proposed hypothesis presented in Section 3.2 was confirmed. Evaluation was performed by demonstrating the practicality of the proposed platform through an externally defined case study and by assessing the fulfilment of the main objectives stated in Section 3.2.

In Section 9.2 the impact of the proposed approach in the MDE community was assessed in terms of publicity, related publications and internal and external uses of Epsilon.

In Section 9.3 the reuse achieved by the layered nature of the architecture was measured in terms of lines of code and the results demonstrated that significant amount of reuse (80%) accross the execution engines and development tools of task-specific model management languages.

10.4 Further Work

Several directions to further work have been identified as a result of this work. One direction is to investigate the need for additional model management tasks which can benefit from a task-specific language. Potential candidates in this category are model simulation and model synchronization.

In terms of the Epsilon prototype, as discussed in Section 9.5, its design and implementation have been driven predominantly by modifiability and
extensibility. However, this design decision has affected performance significantly. Thus, to enable large-scale industrial use, a direction for further work is to rework the tool-support with performance and scalability as primary concerns. To this end, compilers should be used instead of interpreters to delegate computationally expensive operations such as type and variable resolution to the compile-time - instead of the runtime at which they are performed in the current implementation.

Another interesting issue is to provide support in Epsilon for additional upcoming modelling technologies such as the Microsoft Domain Specific Languages Toolkit [49].

This work also raises issues to be investigated in a broader context. Through this work it has been shown that an integrated family of programming languages can solve a wide range of problems by providing customized support to the requirements of each problem without however sacrificing much-desired qualities such as reuse, uniformity and interoperability. It would be of interest to investigate the usefulness of a similar approach to non-model-based software development with a family of inherently reusable and interoperable languages which address different software development tasks (e.g. data persistence and management, object management, presentation and deployment).
Appendix A

Overview of the UML/MOF Class Diagram Visual Notation

This appendix provides a short overview¹ of the notation used to represent UML class models and MOF metamodels for readers that are not familiar with object-oriented modelling. Figure A.1 displays a class diagram that can represent - in different contexts - both a UML model and a MOF metamodel.

A.1 Classes, Attributes and Operations

The model specifies four classes named Graph, GraphElement, Node and Edge via boxes. The first compartment of each class-box contains the name of the class. The second and third compartments contain the attributes and the operations of the class respectively. Thus, the GraphElement class specifies a label attribute of type String and the Edge class specifies a weight attribute of type Integer. Also, the Node class specifies a parameter-less visit() operation.

¹A complete discussion on the semantics of UML class diagrams can be found in [42].
A.2 Associations and References

All lines between two boxes that do not end to a white arrowhead represent associations between the respective classes. Each association has two ends (aka references). Each end optionally specifies a name and a multiplicity. Examples of multiplicities are:

- \( n \): Exactly \( n \)
- \( * \): Zero or more
- \( + \): One or more
- \( n..m \): \( n \) to \( m \) (inclusive)
• n..* : n or more

Wherever the multiplicity is not explicit it is assumed to be exactly one.

Two different types of associations are used: containment (aka composition) and non-containment. Containment associations are denoted by a black diamond at the side of the container. All other associations are assumed to be non-containment. The semantics of the containment relationship is that a contained instance can only be contained in one container and that if the container is deleted, so are the instances it contains.

In the example of Figure A.1 an instance of GraphElement can only be contained in one instance of Graph, and if the container Graph is deleted from the model, so will the contained instances (elements) of GraphElement.

There are also two examples of non-containment associations. The source and target associations between the Node and Edge class specify that each Edge can have exactly one Node as source and exactly one Node as target (implicit multiplicities). Since these associations are not containment, deleting an Edge does not automatically delete the instances of Node that connect to it.

A.3 Inheritance and Abstract Classes

Lines with a white arrowhead between two class-boxes denote an inheritance (subtype) relationship between the respective classes, the class on the side of the arrowhead being the super-type. Both UML and MOF support multiple inheritance and therefore a class can inherit from more than one classes. Classes the name of which appears in italics are abstract; i.e. not instantiable. For example, the GraphElement in the example is abstract and inherited by the Node and Edge classes. Combined with the elements association from Graph to GraphElement, it means that the elements property of Graph can contain both instances of Edge and instances of Node.

\(^2\)UML 2.0 Infrastructure, Final adopted specification, pp 121
A.4 Profiles and Stereotypes

As UML is a general purpose modelling language it needs to provide a specialization mechanism for expressing domain-related concepts where this is necessary. This mechanism is called a profile. In versions 1.x of UML a profile consisted of a set of stereotypes and respective tagged values that could be attached to any model element to specialize it or record additional information. In versions 2.x the profiling mechanism has evolved to allow proper extension of the UML metamodel - a topic which is out of the scope of this discussion. In the examples used in this thesis, the 1.x profiling mechanism is considered. With regard to concrete (graphical) syntax, stereotypes are represented by their name enclosed in $<<, >>$. For instance, in Figure A.2 the table stereotype is attached to the Customer UML class to denote that what is modelled is conceptually a database table, instead of a standard object-oriented class, and the column stereotype is attached to the name attribute to specify that it represents a table column.

<table>
<thead>
<tr>
<th>&lt;&lt;table&gt;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer</td>
</tr>
<tr>
<td>&lt;&lt;column&gt;&gt;-name:String</td>
</tr>
</tbody>
</table>

Figure A.2: Exemplar use of UML stereotypes
Appendix B

Concrete Syntaxes

This appendix provides the complete ANTLR (v2.7.5) grammar specifications that depict the concrete syntax of the model management languages presented in Sections 5.2 and 6.1 – 6.5.

Listing B.1: EOL Concrete Syntax

```java
header {
package org.epsilon.eol.parse;
import java.util.*;
import org.epsilon.commons.parse.problem.*;
import org.epsilon.eol.parse.ast.*;
}

/**
 * Epsilon Object Language Parser
 *
 * Author: Dimitrios S. Kolovos (dkolovos@cs.york.ac.uk)
 * Version: 1.2
 */

class EolLexer extends Lexer;

options {
    k = 3;
    exportVocab = EolParser;
}
```
charVocabulary = '\0' .. '\u00FF';
testLiterals = false;
}
tokens {
  LIT_STRING = "String";
  LIT_INTEGER = "Integer";
  LIT_REAL = "Real";
  LIT_BOOLEAN = "Boolean";
  LIT_MAP = "Map";
  LIT_BAG = "Bag";
  LIT_SEQUENCE = "Sequence";
  LIT_SET = "Set";
  LIT_COLLECTION = "Collection";
  LIT_ORDEREDSET = "OrderedSet";
  LIT_ANY = "Any";
  LIT_NATIVE = "Native";
  NEW = "new";
  VAR = "var";
  RETURN = "return";
  ASYNC = "async";
  DELETE = "delete";
  THROW = "throw";
  IMPORT = "import";
  BREAK = "break";
  BREAKALL = "breakAll";
  FOR = "for";
  TRANSACTION = "transaction";
  ABORT = "abort";
  WHILE_KW = "while";
  IF_KW = "if";
  ELSE_KW = "else";
  OPERATION = "operation";
  MODEL = "model";
  ALIAS = "alias";
  GROUP = "group";
  AS = "as";
  NOT = "not";
  AND = "and";
  XOR = "xor";
}
private ArrayList<ParseProblem> problems = new ArrayList();
protected StringBuffer consumed = new StringBuffer();

public void consume() throws CharStreamException{
    consumed.append(LA(1));
    super.consume();
}

public boolean isKeyword(String keyword) throws CharStreamException{
    if (!isKeywordBoundary(LA(1))) return false;
    if (consumed.length() <= keyword.length()) return true;
    return isKeywordBoundary(consumed.charAt(consumed.length() - keyword.length() - 1));
}

public List<ParseProblem> getParseProblems(){
    return problems;
}

public void reportError(RecognitionException ex) {
    problems.add(new ParseProblem(ex, ParseProblem.ERROR));
}

public void reportError(String s) {
    problems.add(new ParseProblem(s, ParseProblem.ERROR));
}

public void reportWarning(String s) {
    problems.add(new ParseProblem(s, ParseProblem.WARNING));
}
public boolean isKeywordBoundary(char c){
    return !(
        c == '.' ||
        c == '!' ||
        c == '#' ||
        (Character.isJavaIdentifierPart(c) &&
            ((int)c) != 0
        )
    );
}

protected
NL
: ( '\r' '\n'
| '\n' '\r' //Improbable
| '\r'
| '\n'
)
{newline();}
;

WS
: ( ''
| TAB
| NL
| NL
)
{ setType(Token.SKIP); };

TAB
: '\t'
{setColumn(getColumn()-tabsize + 1);}
COMMENT
: "--n"
   (~ ( ' \t'

| ' \n'

))*)
  ((LA(1) != EOF_CHAR)? NL)?

{ $setType(Token.SKIP); }
;

ANNOTATION
: "@"
   (~ ( ' \t'

| ' \n'

))*)
  ((LA(1) != EOF_CHAR)? NL)?

;

LPAREN
options {
    paraphrase = "a left parenthesis ('(')";
}
: '

';

RPAREN
options {
    paraphrase = "a right parenthesis (')')";
}
: ')

';

LSQUARE
options {
    paraphrase = "a left squared parenthesis ('[')";
}
: '

';

RSQUARE

302
options {
    paraphrase = "a right squared parenthesis (']')";
};
'
;

LCURLY
options {
    paraphrase = "{";
};
'{
;

RCURLY
options {
    paraphrase = "}";
};
'}
;

PIPE
options {
    paraphrase = "a pipe ('|')";
};
'|'
;

POINT
options {
    paraphrase = "'.'";
};
'.
;

POINT_POINT
options {
    paraphrase = "..";
};
"..
;
COMA
options {
  paraphrase = "','";
};
',';

SEMI
options {
  paraphrase = "';'";
};
';'
;
QUESTIONMARK
options {
  paraphrase = "'?'";
};
'?'
;
DOLLAR
options {
  paraphrase = "'$'";
};
'$'
;
/
AROBAPRE
options {
  paraphrase = "@pre";
};
"@pre"
;
*/

DIESE
<table>
<thead>
<tr>
<th><strong>options</strong></th>
<th><strong>paraphrase</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>&quot;#&quot;</td>
</tr>
<tr>
<td>:</td>
<td>&quot;:&quot;</td>
</tr>
<tr>
<td>=</td>
<td>&quot;=&quot;</td>
</tr>
<tr>
<td>::</td>
<td>&quot;::&quot;</td>
</tr>
<tr>
<td>::</td>
<td>&quot;::&quot;</td>
</tr>
<tr>
<td>-&gt;</td>
<td>&quot;-&gt;&quot;</td>
</tr>
</tbody>
</table>
"->";

STAR
options {
  paraphrase = "*";
};
  
  
DIV
options {
  paraphrase = "/";
};
  
  
PLUS
options {
  paraphrase = "+";
};
  
  
MINUS
options {
  paraphrase = "-";
};
  
  
EQ
options {
  paraphrase = "=";
};
  
  
NE
options {

paraphrase = "<>";
}
"<>"
;

LT
options {
    paraphrase = "<";
}
'<'
;

LE
options {
    paraphrase = "<=";
}
"<=";

GT
options {
    paraphrase = ">";
}
'>'
;

GE
options {
    paraphrase = ">=";
}
">="
;

protected
DIGIT
: '0'..'9'
;
EXC
  : 
    `'!'`
    ;

protected
ALPHA
  : 'a'..'z'
  | 'A'..'Z'
  | '_'
  // For Unicode compatibility (from 0000 to 00ff)
  | '\u00C0' .. '\u00D6'
  | '\u00D8' .. '\u00F6'
  | '\u00F8' .. '\u00FF'
    ;

INT
options {
  paraphrase = "a number";
} : (DIGIT)+
  (('.'DIGIT)=> '.') (DIGIT)+ {$setType(FLOAT);}}?
    ;

protected
ESC
  : `\\'`
    ( 'n' {$setText("\n");}
    | 'r' {$setText("\r");}
    | 't' {$setText("\t");}
    | 'b' {$setText("\b");}
    | 'f' {$setText("\f");}
    | '"' {$setText("\"");}
    | '\\\' {$setText("\\");}
    )
    ( '0'..'3')
    
    options {
      warnWhenFollowAmbig = false;
    }
String s = $getText;
int i;
int ret = 0;
String ans;
for (i=0; i<s.length(); ++i)
    ret = ret*8 + s.charAt(i) - '0';
ans = String.valueOf((char) ret);
$setText(ans);
STRING_NAME
options {
    paraphrase = "a string name";
}
: '"'!
    ( ESC
        | ¬("\"|\""")
    )*
    '"'!
($setType(NAME);)
;
/*
STRING_NAME
options {
    paraphrase = "a string name";
}
: '"'!
    NAME
    '"'!
($setType(NAME);)
;
*/
RECOVER
: ¬( '0'..'9' | 'A'..'Z' | 'a'..'z' | '_'
    | '+'|'-'|'*'|'/'
    | '>='< | '='
    | '\\}|\}|\}|';';';';';'
    | '{'}|'{'}|'{'}|'{'}|'{'}
    | '\t'|'\n'|'\r'
    | '\u00C0' .. '\u00D6'
    | '\u00D8' .. '\u00F6'
    | '\u00F8' .. '\u00FF'
)
    ( ¬( ' '
        | '\t'
        | '\n'
        | '\r'
    )
))∗
\{
    String s=$getText;
    $setText("line " + getLine() + ": " + s + ": Unknown lexem; Token ignored");
    reportWarning($getText);
    $setType(Token.SKIP);
\}

NAME
options { 
    paraphrase = "a name";
    testLiterals = true;
} : (’˜’)? (ALPHA) (ALPHA | DIGIT)*;

class EolParser extends Parser;

options {
    k = 6;
    exportVocab = EolParser;
    buildAST = true;
    ASTLabelType = "EolAst";
}
tokens { 
    CONSTR;
    CLASSCONTEXT;
    OPCONTEXT;
    CONSTR_BODY;
    STEREOTYPE;
    FORMAL;
    PARAMLIST;
    TYPELIST;
    LET;
    ASSIGN_STATEMENT;
    SPECIAL_ASSIGN_STATEMENT;
    QUALIFIERS;
    PARAMETERS;
    PARAMETERSDEFININGVARS;
    DECLARATION;
    VALUE;
EXPRLIST;
EXPRRANGE;
OPERATOR;
TYPE;
TYPEINIT;
ENUMERATION;
ENUMERATION_VALUE;
COLLECTION;
NATIVE;
BOOLEAN;
FEATURECALL;
BLOCK;
IF;
THEN;
ELSE;
WHILE;
VAR;
RETURN;
ASYNC;
DELETE;
THROW;
BREAK;
FOR;
TRANSACTION;
ABORT;
HELPERMETHOD;
EOLMODULE;
IO;
NAMELIST;
PATHNAME;
IMPORT;
CONTINUE;
BREAKALL;
EXECUTABLEANNOTATION;
ANNOTATIONBLOCK;
private ArrayList<ParseProblem> problems = new ArrayList();

public List<ParseProblem> getParseProblems() {
    return problems;
}

public void reportError(RecognitionException ex) {
    problems.add(new ParseProblem(ex, ParseProblem.ERROR));
}

public void reportError(String s) {
    problems.add(new ParseProblem(s, ParseProblem.ERROR));
}

public void reportWarning(String s) {
    problems.add(new ParseProblem(s, ParseProblem.WARNING));
}
}

```eolModule
:
(importStatement)*
(modelDeclaration | annotationBlock)*
block
{
    helperMethod|
    annotationBlock
}*
{
    #eolModule = #([EOLMODULE,"Eol module"], #eolModule);
    ;
}

helperMethod
: OPERATION" (t:typeName#{t.setType(TYPEINIT);})? name:NAME LPAREN! (formalParameterList)? RPAREN! (POINTS! r:typeName#{r.setType(TYPEINIT);})? LCURLY! block RCURLY!
{
    #helperMethod.setType(HELPERMETHOD);
    ;
}
formalParameterList
  : formalParameter (COMA! formalParameter)*
  {
    formalParameterList =
      #([PARMLIST,"Parameter list"], formalParameterList);
  }

formalTypeList
  : typeName (COMA! typeName)*
  {
    formalTypeList =
      #([TYPETLIST,"Type list"], formalTypeList);
  }

formalParameter
  : NAME POINTS` (#POINTS.setType(FORMAL);
    #POINTS.setText("Formal parameter");) typeName
 ;

block
  : (statement)*
  {
    block = #([BLOCK,"{...}"], block);
  }

blockOrExpression
  : (LCURLY! block RCURLY!)
  | POINTS! logicalExpression
  ;

executableAnnotation
  : DOLLAR` NAME logicalExpression
  {
    executableAnnotation.setType(EXECUTABLEANNOTATION);
  }

annotationBlock
  : (ANNOTATION|executableAnnotation)+
  {
    annotationBlock = #([ANNOTATIONBLOCK,"@..."] , annotationBlock);
  }
classifierType
  : pathName
   
   {#classifierType.setType(TYPE);}

; 

varExpression
  : VAR^ NAME (POINTS! (NEW!
    {#varExpression.setText("new");})?
    x:typeName {#x.setType(TYPEINIT);})?;

newExpression
  : NEW^ x:typeName {#x.setType(TYPEINIT);};;

returnStatement
  : RETURN^ (logicalExpression)? SEMI!
    {#returnStatement.setType(RETURN);}

; 

asyncStatement
  : ASYNC^ (logicalExpression)? SEMI!
    {#asyncStatement.setType(ASYNC);}

; 

deleteStatement
  : DELETE^ (logicalExpression)? SEMI!
    {#deleteStatement.setType(DELETE);}

; 

throwStatement
  : THROW^ (logicalExpression)? SEMI!
    {#throwStatement.setType(THROW);}

; 

importStatement
  : IMPORT^ (STRING|NAME) SEMI!
    {#importStatement.setType(IMPORT);}

;
breakStatement
  : BREAK` SEMI!
    {#breakStatement.setType(BREAK);}  
  ;

breakAllStatement
  : BREAKALL` SEMI!
    {#breakAllStatement.setType(BREAKALL);}  
  ;

continueStatement
  : CONTINUE` SEMI!
    {#continueStatement.setType(CONTINUE);}  
  ;

statement
  : (logicalExpression) ((ASSIGNMENT` logicalExpression)|
    (SPECIAL_ASSIGNMENT` logicalExpression))? SEMI!
    | returnStatement
    | asyncStatement
    | throwStatement
    | breakStatement
    | abortStatement
    | breakAllStatement
    | continueStatement
    | forStatement
    | transactionStatement
    | ifStatement
    | whileStatement
    | deleteStatement
  ;

logicalExpression
  : relationalExpression
    (("and" | "or" | "xor" | "implies")
    relationalExpression
    (#logicalExpression.setType(OPERATOR));))*
  ;

relationalExpression
additiveExpression : additiveExpression ((EQˆ|GTˆ|LTˆ|GEˆ|LEˆ|NEˆ)
additiveExpression ({#relationalExpression.setType(OPERATOR);})? ;

additiveExpression : multiplicativeExpression ((PLUSˆ|MINUSˆ)
multiplicativeExpression ({#additiveExpression.setType(OPERATOR);})*) ;

multiplicativeExpression : unaryExpression ((STARˆ|DIVˆ) unaryExpression
{#multiplicativeExpression.setType(OPERATOR);})*) ;

unaryExpression : ((NOTˆ|MINUSˆ)
{#unaryExpression.setType(OPERATOR);})? postfixExpression ;

postfixExpression : primaryExpression((POINTˆ|ARROWˆ) fc:featureCall
{#fc.setType(FEATURECALL);})* ;

primaryExpression : literalCollection | literal | featureCall | LPAREN! logicalExpression RPAREN! | varExpression | newExpression ;

literalCollection : collType:oclCollection! LCURLY!
(collDef:expressionListOrRange!)? RCURLY!
{#literalCollection = #{collType, collDef};}
expressionListOrRange
: logicalExpression
  ( (#expressionListOrRange =
    (#([EXPRLIST,"Expression list"], #expressionListOrRange);}
   | (COMA! logicalExpression)+
   (#expressionListOrRange =
    (#([EXPRLIST,"EXPRLIST"], expressionListOrRange);}
   | POINT_POINT! logicalExpression
   (#expressionListOrRange = #(EXPRRANGE,"EXPRRANGE"],
    expressionListOrRange);}
  )
)
;

litteral
: number
;

ifStatement
: IF_KW! logicalExpression L CURLY! block
  RCURLY! (ELSE_KW! L CURLY! block RCURLY!)?
  (#ifStatement = #([IF,"if"], ifStatement);}
);

whileStatement
: WHILE_KW! logicalExpression L CURLY! block RCURLY!

(#whileStatement = #([WHILE,"while"], whileStatement);}
);

forStatement
: FOR! LPAREN! (NAME|formalParameter)
  IN! logicalExpression RPAREN! L CURLY! block RCURLY!

(#forStatement.setType(FOR);)
;
transactionStatement
  : TRANSACTION" (NAME (COMA! NAME)*)? LCURLY! block RCURLY!

  {
  #transactionStatement.setType(TRANSACTION);
  
  
}

abortStatement
  : ABORT" (logicalExpression)? SEMI!
  {
  #abortStatement.setType(ABORT);
  
  
}

qualifiers
  : LSQUARE! actualParameterList RSQUARE!
  {
  
  #qualifiers.setType(QUALIFIERS);
  #qualifiers.setText("Qualifiers");
  
  
}

featureCall {
  boolean previous = false;

  
  : pn:pathName!
  ((LSQUARE NAME RSQUARE)=>(qualifiers))?
  (AROBAPRE! {previous = true;})?
  ( qualifiers
  | parameters
  )?

  {
  
  #featureCall = #(pn, #featureCall);
  if (previous)
    #featureCall = #(#[AROBAPRE, "AROBAPRE"], #featureCall);
  
  
}

parameters {
  int i = 1; boolean b = false;

  
  : LPAREN! {
  
  int lparen = 0;

  

  

while ((lparen != 0) || (LA(i) != EOF) && (! b) && (LA(i) != RPAREN)) {
    b = (LA(i) == PIPE);
    if (LA(i) == LPAREN)
        lparen++;
    else if (LA(i) == RPAREN)
        lparen--;
    ++i;
}

{b}? declarator (logicalExpression)? | actualParameterList
RPAREN!
{
    if (b)
        #parameters = #([PARAMETERSDEFININGVARS,
            "Parameters defining variables"], parameters);
    else {
        #parameters.setText("Parameters");
        #parameters.setType(PARAMETERS);
    }
}

declarator
: declaration
    (SEMI! declaration)*
    PIPE!
    {#declarator = #([DECLARATION,"Declarator"],
        #declarator);}

modelDeclaration
:
    MODEL` nameList (ALIAS! nameList)? (POINTS! NAME)? SEMI!

groupDeclaration
GROUP` nameList AS! NAME SEMI! ;

nameList
: (NAME (COMA! NAME)*)?;
{nameList = #(NAMELIST,"Name list"], nameList);} ;
declaration
: NAME
 (COMA! NAME)*
 (POINTS! t:typeName {t.setType(TYPEINIT);})?
 (EQ logicalExpression)?
{#declaration = #[DECLARATION,"Declaration"], #declaration};
;
actualParameterList
: (logicalExpression (COMA! logicalExpression)*)?
{#actualParameterList = #[EXPRLIST, "Expression list"], actualParameterList};
;
typeName
: pathName {typeName.setType(TYPE);} |
enumType |
oclType ;
enumType
: "enum"! LCURLY! DIESE! NAME
 (COMA! DIESE! NAME)\+ RCURLY!
{enumType = #[TYPE,"TYPE"],
 #([ENUMERATION,"Enumeration"], #enumType)};
;
oclType
: ( LIT_REAL
| LIT_INTEGER
| LIT_STRING
| LIT_BOOLEAN
| LIT_MAP
| LIT_ANY
| oclCollection
| nativeType

{#oclType.setType(TYPE);}

nativeType
: LIT_NATIVE" LPAREN! STRING RPAREN!
{#nativeType.setType(NATIVE);}

oclCollection
: (LIT_SET|
  LIT_SEQUENCE|
  LIT_ORDEREDSET|
  LIT_BAG|
  LIT_COLLECTION)
(LPAREN! type:typeName RPAREN!)?
{#oclCollection.setType(COLLECTION);}

pathName
: (metamodel:NAME! EXC!)? head:NAME
 (FOUR_POINTS! field:NAME! { head.setText(head.getText()
    + ":" + field.getText();
  })
  (DIESE! label:NAME!)*
  { if (metamodel == null)
Listing B.2: ECL Concrete Syntax

header {
package org.epsilon.ecl.parse;
import java.util.*;
import org.epsilon.commons.parse.problem.*;
import org.epsilon.eol.parse.ast.*;
}

/**
 * Epsilon Comparison Language Parser
 * @author Dimitrios S. Kolovos (dkolovos@cs.york.ac.uk)
 * @version 1.1
 */

if (label != null) {
    #pathName.setText(#pathName.getText() + "#" + label.getText());
    #pathName.setType(ENUMERATION_VALUE);
} else
    #pathName.setText(head.getText());

number
    : INT
    | FLOAT
    | STRING
    | bool
    | oclType

bool
    : ( TRUE | FALSE)
/#{bool.setType(BOOLEAN);}
class EclLexer extends EolLexer;

options {
    exportVocab = EclLexer;
    charVocabulary = '\0' .. '\u00FF';
}
tokens {
    RULE = "rule";
    EXTENDS = "extends";
    COMPARE = "compare";
    DO = "do";
    PRE = "pre";
    POST = "post";
    MATCH = "match";
    TRANSFORM = "transform";
    GUARD = "guard";
    WITH = "with";
}

EXC :
    '!' 
    ;

class EclParser extends EolParser;

options {
    importVocab = EclLexer;
    exportVocab = EclParser;
    buildAST = true;
    ASTLabelType = "EolAst";
}
tokens {
    MATCHRULE;
    PREBLOCK;
    POSTBLOCK;
    COMPAREBLOCK;
eclModule:
  (importStatement)*
  (modelDeclaration | groupDeclaration | preBlock | postBlock | matchRule | helperMethod | annotationBlock) *
  {#eclModule = #([ECLMODULE,"Ecl Module"], eclModule);} ;

matchRule:
  RULE^ name:NAME
  MATCH! left:formalParameter
  WITH! right:formalParameter
  (superRules)?
  LCURLY! (guardBlock)? (compareBlock)? (doBlock)? RCURLY!

  {#matchRule.setType(MATCHRULE);} ;

superRules:
  : EXTENDS^ NAME (COMA! NAME)*
  {superRules.setType(SUPERRULES);} ;

guardBlock:
  : GUARD^ blockOrExpression
  {guardBlock.setType(GUARDBLOCK);} ;

preBlock:
  : PRE^ (NAME)? LCURLY! (block)? RCURLY!
Listing B.3: ETL Concrete Syntax

```java
package org.epsilon.etl.parse;
import java.util.*;
import org.epsilon.commons.parse.problem.*;
import org.epsilon.eol.parse.ast.*;
import org.epsilon.eol.parse.*;

class EtlLexer extends EolLexer;

options {
  exportVocab = EtlLexer;
  charVocabulary = '\0' .. '\u00FF';
}
```

class EtlParser extends EolParser;

options {
  importVocab = EtlLexer;
  exportVocab = EtlParser;
  buildAST = true;
  ASTLabelType = "EolAst";
  k = 5;
}

tokens {
  PREBLOCK;
  POSTBLOCK;
  ETLMODULE;
}

etlModule :
  (importStatement)*
Listing B.4: EVL Concrete Syntax

```java
{modelDeclaration | preBlock | transformRule | postBlock | helperMethod | annotationBlock}*

{"etlModule = #{ETLMODULE,"Etl Module"}, etlModule};

transformRule

: RULE! name:NAME
TRANSFORM source:formalParameter
TO! target:formalParameterList
(superRules)?
LCURLY! (guardBlock)? (block) RCURLY!
;

superRules

: EXTENDS" NAME (COMA! NAME)*
;

guardBlock

: GUARD" blockOrExpression
;

preBlock

:
PRE" (NAME)? LCURLY! (block)? RCURLY!
{"preBlock.setType(PREBLOCK);}
;

postBlock

:
POST" (NAME)? LCURLY! (block)? RCURLY!
{"postBlock.setType(POSTBLOCK);}
;

header {
package org.epsilon.evl.parse;
import java.util.*;
```
import org.epsilon.commons.parse.problem.*;
import org.epsilon.eol.parse.ast.*;
}

/**
 * Epsilon Validation Language Parser
 * @author Dimitrios S. Kolovos (dkolovos@cs.york.ac.uk)
 * @version 1.1
 */
class EvlLexer extends EolLexer;

options {
    exportVocab = EvlLexer;
    charVocabulary = '\0' .. '\u00FF';
}

tokens {
    CONTEXT = "context";
    CONSTRAINT = "constraint";
    CRITIQUE = "critique";
    GUARD = "guard";
    MESSAGE = "message";
    FIX = "fix";
    DO = "do";
    CHECK = "check";
    TITLE = "title";
    TYPEOF = "typeof";
    KINDOF = "kindOf";
    HIGH = "high";
    MEDIUM = "medium";
    LOW = "low";
    PRE = "pre";
    POST = "post";
}

EXC
:
class EvlParser extends EolParser;

options {
    importVocab = EvlLexer;
    exportVocab = EvlParser;
    buildAST = true;
    ASTLabelType = "EolAst";
}

tokens{
    EVLMODULE;
    CONTEXT;
    CONSTRAINT;
    FIX;
    TITLE;
    DO;
    CHECK;
    MESSAGE;
    CRITIQUE;
    GUARD;
    PREBLOCK;
    POSTBLOCK;
}

evlModule
    : (importStatement)* (preBlock |postBlock | context | helperMethod | annotationBlock) *
    {#evlModule = #([EVLMODULE,"Evl Module"], evlModule);} ;

context
    :
    CONTEXT" (NAME (TYPEOF|KINDOF))? typeName
    LCURLY! (guard)?
    (constraint|critique|annotationBlock) * RCURLY! ;

330
constraint
  :
  CONSTRAINT` name:NAME! 
  LCURLY! (guard)? constraintBody (message)?
  (fix)* RCURLY!
  {#constraint.setText(name.getText());}
  ;

critique
  :
  CRITIQUE` (HIGH|MEDIUM|LOW)? name:NAME! 
  LCURLY! (guard)? constraintBody (message)?
  (fix)* RCURLY!
  {#critique.setText(name.getText());}
  ;

constraintBody :
  CHECK` (LCURLY! block RCURLY! | 
  POINTS! logicalExpression);

message : MESSAGE` (LCURLY! block RCURLY! | 
  POINTS! logicalExpression );

guard : GUARD` (LCURLY! block RCURLY! | 
  POINTS! logicalExpression );

fix : FIX` (LCURLY! title fixBody RCURLY!);

title : TITLE` (LCURLY! block RCURLY! | 
  POINTS! logicalExpression );

fixBody : DO` LCURLY! block RCURLY!;

preBlock
  :
  PRE` (NAME)? LCURLY! (block)? RCURLY!
  {#preBlock.setType(PREBLOCK);} 
  ;

postBlock

331
Listing B.5: EWL Concrete Syntax

header {
  package org.epsilon.ewl.parse;
  import java.util.*;
  import org.epsilon.commons.parse.problem.*;
  import org.epsilon.eol.parse.ast.*;
}
/**
 * Epsilon Wizard Language Parser
 * @author Dimitrios S. Kolovos (dkolovos@cs.york.ac.uk)
 * @version 1.1
 *
*/
class EwlLexer extends EolLexer;

options {
  exportVocab = EwlLexer;
  charVocabulary = `\0' .. `\u00FF';
}
tokens {
  EWLMODULE = "ewlModule";
  WIZARD = "wizard";
  GUARD = "guard";
  DO = "do";
  TITLE = "title";
  CHECK = "check";
}
EXC
  ;
class EwlParser extends EolParser;

options {
    importVocab = EwlLexer;
    exportVocab = EwlParser;
    buildAST = true;
    ASTLabelType = "EolAst";
}

tokens{
    EWLMODULE;
}

ewlModule :
    (importStatement)* (wizard | helperMethod | annotationBlock) *
    {#ewlModule = #([EWLMODULE,"Ewl Module"], ewlModule);}
;

wizard :
    WIZARD^ name:NAME!
    LCURLY! (guard)? title wizardBody RCURLY!
    {#wizard.setText(name.getText());}
;

guard :
    GUARD^ (LCURLY! block RCURLY! |
    POINTS! logicalExpression );

title :
    TITLE^ (LCURLY! block RCURLY! |
    POINTS! logicalExpression );

check :
    CHECK^ (LCURLY! block RCURLY! |
    POINTS! logicalExpression );

wizardBody : DO^ LCURLY! block RCURLY!;
Listing B.6: EML Concrete Syntax

```java
header {
    package org.epsilon.eml.parse;
    import java.util.*;
    import org.epsilon.commons.parse.problem.*;
    import org.epsilon.eol.parse.ast.*;
}

/**
 * Epsilon Merging Language Parser
 *
 * @author Dimitrios S. Kolovos (dkolovos@cs.york.ac.uk)
 * @version 1.2
 *
 */
class EmlLexer extends EtlLexer;

options {
    exportVocab = EmlLexer;
    charVocabulary = '\0' .. '\u00FF';
    testLiterals = false;
}

tokens {
    INTO = "into";
    MERGE = "merge";
    WITH = "with";
}

EXC
:
    '!' ;

class EmlParser extends EtlParser;

options {
    exportVocab = EmlParser;
    importVocab = EmlLexer;
}
```

334
buildAST = true;
ASTLabelType = "EclAst";
}
tokens{
PREBLOCK;
POSTBLOCK;
ETLMODULE;
EMLMODULE;
}
emlModule :
(importStatement)*
(preBlock | postBlock | mergeRule | transformRule |
helperMethod | annotationBlock) *
{"emlModule = #{[EMLMODULE,"Eml Module"], emlModule};} 
;
mergeRule :
RULE! name:NAME
MERGE! left:formalParameter
WITH! right:formalParameter
INTO! target:formalParameterList
(superRules)?
LCURLY! (guardBlock)? (block) RCURLY!
;
Appendix C

List of Acronyms

A
ATL: Atlas Transformation Language

D
DSL: Domain Specific Language

E
ECL: Epsilon Comparison Language
EGL: Epsilon Generation Language
EMC: Epsilon Model Connectivity
EML: Epsilon Merging Language
EMF: Eclipse Modeling Framework
EOL: Epsilon Object Language
Epsilon: Extensible Platform of Integrated Languages for Model Management
ETL: Epsilon Transformation Language
EVL : Epsilon Validation Language

G

GMF : Graphical Modeling Framework

M

MDA : Model Driven Architecture
MDE : Model Driven Engineering
MOF : Meta Object Facility

O

oAW : Open ArchitectureWare
OMG : Object Management Group

Q

QVT : Queries/Views/Transformations

T

TMF : Textual Modeling Framework

U

UML : Unified Modeling Language
X

XML : Extensible Markup Language
Bibliography


ECOOP Workshop on Domain-Specific Program Development (DSPD 2006), Nantes, France, July 2006.


345


[156] Pascual Queralt, Luis Hoyos, Artur Boronat, Jose A. Carsi e Isidro Ramos. Un motor de transformacion de modelos con soporte para el lenguaje QVT relations. In III Taller sobre Desarrollo de Software Dirigido por Modelos. MDA y Aplicaciones (DSDM’06), 2006.


