Structures and Processes for Managing Model-Metamodel Co-evolution

Louis Mathew Rose

This thesis is submitted in partial fulfilment
of the requirements for the degree of
Doctor of Philosophy.

University of York,
York,
YO10 5DD

Department of Computer Science

October 2010

Abstract

Software changes over time. During the lifetime of a software system, unintended behaviour must be corrected and new requirements satisfied. Because software changes are costly, tools for automatically managing change are commonplace. Contemporary development environments can automatically perform change management tasks such as impact analysis, refactoring and background compilation.

Increasingly, models and modelling languages are first-class citizens in software development. Model-Driven Engineering (MDE), a state-of-the-art approach to software engineering, prescribes the use of models throughout the software engineering process and uses automated transformations to generate code from models.

Contemporary MDE environments provide little support for managing a type of evolution termed *model-metamodel co-evolution*, in which changes to a modelling language are propagated to models. This thesis demonstrates that model-metamodel co-evolution occurs often in MDE projects, and that dedicated structures and processes for its management increase the productivity and understandability of the development process. Structures and processes for managing model-metamodel co-evolution are proposed, developed, and then evaluated by comparison to existing structures and processes with quantitive and qualitative techniques.

For my Nanna Spence

Contents

List of Figures

List of Tables

Acknowledgements

To be completed.

Author Declaration

Except where stated, all of the work contained in this thesis represents the original contribution of the author. Section 6.3 includes work conducted in collaboration with others and the contributions of the thesis author are made clear in that section.

Parts of the work described in this thesis have been previously published by the author in:

- **The Epsilon Generation Language**, Louis M. Rose and Richard F. Paige and Dimitrios S. Kolovos and Fiona A.C. Polack in *Proc. European Conference on Model Driven Architecture – Foundations and Applications (ECMDA-FA)*, volume 5095 of LNCS, pages 1-16. Springer, 2008.
- **Constructing Models with the Human-Usable Textual Notation**, Louis M. Rose and Richard F. Paige and Dimitrios S. Kolovos and Fiona A.C. Polack in *Proc. International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 5301 of LNCS, pages 249-263. Springer, 2008.
- **An Analysis of Approaches to Model Migration**, Louis M. Rose and Richard F. Paige and Dimitrios S. Kolovos and Fiona A.C. Polack in *Proc. Joint Model-Driven Software Evolution and Model Co-evolution and Consistency Management (MoDSE-MCCM) Workshop*, co-located with MoDELS 2009.
- **Enhanced Automation for Managing Model and Metamodel Inconsistency**, Louis M. Rose and Dimitrios S. Kolovos and Richard F. Paige and Fiona A.C. Polack in *Proc. International Conference on Automated Software Engineering (ASE)*, pages 545-549, ACM Press, 2009.
- **Concordance: An Efficient Framework for Managing Model Integrity**, Louis M. Rose, Dimitrios S. Kolovos, Nicholas Drivalos, James. R. Williams, Richard F. Paige, Fiona A.C. Polack, and Kiran J. Fernandes in *Proc. European Conference on Modelling Foundations and Applications*

(ECMFA), volume 6138 of LNCS, pages 62-73. Springer, 2010.

- **Model Migration with Epsilon Flock**, Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack in *Proc. International Conference on the Theory and Practice of Model Transformations (ICMT)*, volume 6142 of LNCS, pages 184-198. Springer, 2010.
- **Model Migration Case**, Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack in *Proc. Transformation Tools Contest (TTC)*, co-located with TOOLS 2010.
- **Migrating Activity Diagrams with Epsilon Flock**, Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack in *Proc. Transformation Tools Contest (TTC)*, co-located with TOOLS 2010.

In addition, the thesis author has contributed to [Kolovos *et al.* 2007a], [Kolovos *et al.* 2007b] and [Paige *et al.* 2009].

Chapter 1

Introduction

The complexity of new software is increasing over time. Today's software developers build distributed and interoperating systems with sophisticated graphical interfaces rather than the insular, monolithic, and command-line driven mainframe applications built by their predecessors.

Some of the software demanded by users and developers today is so complicated that its construction is not possible, even using state-of-the-art software engineering techniques [Selic 2003]. However, demand does not appear to exceed capability in all areas of computer science.

Hardware development, for example, seems to advance more quickly than software development. Each year, faster personal computers with larger disk drives become available, while operating systems, office software and development environments seem to improve more gradually. [Brooks 1986] suggests that radical advances in software development occur only by raising the level of abstraction at which software is specified.

1.1 Model-Driven Engineering

Historically, raising the level of abstraction of software development has led to increased productivity. For example, assembly language provides mnemonics for machine code, allowing developers to disregard superfluous detail (such as the binary representation of instructions). Object-orientation and functional programming permit further abstraction over assembler, enabling developers to express solutions in a manner that is more representative of their problem domain.

Model-Driven Engineering (MDE) is a contemporary approach to software engineering that seeks to abstract away from technological details (such as programming languages and off-the-shelf software components) and towards the problem domain of the system (for example: accounting, managing patient records, or searching the Internet). To this end, MDE prescribes, throughout the software engineering process, the use of models to capture the relevant

details of the problem domain. Software development is driven by manipulating (transforming, validating, merging, comparing, etc.) the models to automatically generate working code.

MDE reportedly provides many benefits over traditional approaches to software engineering. [Watson 2008] presents results from two unpublished case studies, and suggests that MDE can lead to increased productivity by reducing the amount of time to develop a system, and by reducing the number of defects discovered throughout development. [Kleppe *et al.* 2003] discusses the ways in which MDE can be used to increase the productivity of software development and the maintainability and portability of software systems.

Notwithstanding its benefits, MDE introduces additional challenges for software development. [Kolovos *et al.* 2008c] reports scalability issues with contemporary MDE, noting that large models are commonplace in many software engineering projects and that contemporary MDE environments cannot be used to manipulate large models. [Mens & Demeyer 2007] demonstrate that MDE introduces new challenges for managing change throughout the lifetime of a system. This thesis focuses on the latter challenge, which is part of a branch of computer science termed *software evolution*.

1.2 Software Evolution

Software changes over time. During the lifetime of a software system, unintended behaviour must be corrected and new requirements satisfied. Because modern software systems are rarely isolated from other systems, changes are also made to facilitate interoperability with new systems [Sjøberg 1993].

Software evolution is an area of computer science that focuses on the way in which a software system changes over time in response to external pressures (such as changing requirements, or the discovery of unintended behaviour). The terms software evolution and software maintenance are used interchangeably in the software engineering literature. In this thesis, *evolution* is preferred to *maintenance*, because the latter can imply deterioration caused by repeated use, and most software systems do not deteriorate in this manner [Ramil & Lehman 2000]. Other than sharing some terminology, software evolution is not related to evolutionary algorithms, a branch of computer science that encompasses genetic programming and genetic algorithms.

In the past, studies have suggested that software evolution can account for as much as 90% of a development budget [Erlikh 2000, Moad 1990], and there is no reason to believe that the situation is different today. Although [Sommerville 2006, ch. 21] describes such figures as uncertain, precise figures are not required to demonstrate that the effects of evolution can inhibit the productivity of software development.

For example, suppose that we are developing a software system using a combination of hand-written code and off-the-shelf components. Part way

through development, one of the components changes to support a new requirement. When using the new version of the component, we must first determine whether our system exhibits any unintended behaviour, identify the cause of the unintended behaviour, and change the system accordingly. The resources allocated to correcting any unintended behaviour are not being used to develop features for the users of our system.

Primarily, software evolution research seeks to reduce the cost of making changes to a system. Analysis of the effects of evolution facilitates informed decisions about how best to manage evolution. For example, analysis of our system might indicate that using a new version of a component will introduce three defects, but simplify the implementation of two features. Studying the way in which systems evolve lead to improvements in software development tools and processes that reduce the effects of evolution. For example, contemporary software development environments recognise compilation as a common activity during software evolution, and often perform automatic and incremental compilation of source code in the background. Future changes to a system might be anticipated by identifying the ways in which the system has previously evolved. For example, understanding the ways in which our system has been affected by using a new version of a component might highlight ways in which our system can be better protected against changes to its dependencies.

1.3 Motivation: Software Evolution in MDE

Proponents of MDE suggest that, compared to traditional approaches to software engineering, application of MDE leads to systems that better support evolutionary change [Kleppe *et al.* 2003]. [Frankel 2002] suggests that large-scale systems, developed with traditional approaches to software engineering, are examples of a modern-day Sisyphus[1], whose developers must constantly perform evolution to support conformance to changing standards and interoperability with external systems, and that MDE can be used to reduce the cost of software evolution. However, [Mens & Demeyer 2007] report that MDE introduces additional challenges for managing evolution.

In particular, MDE introduces development artefacts not typically used in traditional software engineering, such as models and modelling languages. For traditional development artefacts (such as source code and documentation), contemporary development environments provide some assistance for performing software evolution activities (by, for example, providing transformations that automatically restructure code). However, there is little support for software evolution activities that involve the additional development artefacts introduced by MDE.

[1] In Greek mythology, Sisyphus was condemned to an eternity of repeatedly rolling a boulder to the top of a mountain, only to see it return to the mountain's base.

There is an urgent need for development environments that support the evolution of models, modelling languages and other MDE development artefacts. Without this support, software development with MDE is less productive and harder to reason about. Resources that could be used to improve the software system are wasted, used instead to manually identify and reconcile the problems caused by evolution. The way in which the system changes over time is not recorded and cannot be analysed to anticipate future changes or to improve the development process.

To this end, this thesis explores the extent to which the productivity and understandability of software development with MDE can be increased by enhancing contemporary development environments with support for software evolution activities involving MDE development artefacts, such as models and modelling languages.

1.4 Research Hypothesis

The research presented in this thesis explores the hypothesis below. The emboldened terms are potentially ambiguous, and their definition follows the hypothesis.

> *In existing MDE projects, the evolution of* ***MDE development artefacts*** *is typically managed in an ad-hoc manner with little regard for re-use. Dedicated structures and processes for* ***managing evolutionary change*** *can be designed by analysing evolution in existing MDE projects. Furthermore, supporting those dedicated structures and processes in contemporary MDE environments is beneficial in terms of increased* ***productivity*** *and* ***understandability*** *of software development.*

In this thesis, the terms below have the following definitions:

MDE development artefacts. Compared to traditional approaches to software engineering, MDE uses additional development artefacts as first-class citizens in the development process. The additional development artefacts particular to MDE include models and modelling languages, as well as model management operations (such as model transformations). Chapter 2 describes models, modelling languages and model management operations in more detail.

Managing evolutionary change. Contemporary computer systems are constructed by combining numerous interdependent artefacts. Evolutionary changes to one artefact can affect other artefacts. For example, changing a database schema might cause data to become invalid with respect to the

database integrity constraints, and changing source code may require recompilation of object code to ensure the latter is an accurate representation of the former. Managing evolutionary change typically comprises three related activities: *identifying* when a change has occurred, *reporting* the effects of a change, and *reconciling* affected artefacts in response to a change.

Productivity is a measure of the amount of work required to complete a development activity. For example, the productivity of data entry might be increased by using an Optical Character Recognition (OCR) system rather than a typist. In this scenario, the extent to which productivity might be increased is affected by at least the following: the accuracy and capabilities of the OCR system, the speed and accuracy of the typist, and the legibility and consistency of the data. In general, productivity is affected by many factors.

Understandability is a measure of the ease with which the requirements, implementation, dependencies, and other qualities of a system can be identified from the representation of that system. For example, the function of a circuit is arguably easier to understand when examining a schematic of the circuit rather than a Printed Circuit Board (PCB) that implements the circuit, because components are often arranged to save space on a PCB. Understanding why, how and when a system has evolved in the past is useful for anticipating future changes and improving the development process. Clearly, understandability is somewhat subjective. Due to differences in knowledge and experience, a representation that is easy to understand for one person may be difficult for another.

1.4.1 Thesis Objectives

The objectives of the thesis are to:

1. Identify and analyse the evolution of MDE development artefacts in existing projects.

2. Investigate the extent to which existing structures and processes can be used to manage the evolution of MDE development artefacts.

3. Propose and develop new structures and processes for managing the evolution of MDE development artefacts, and integrate those structures and processes with a contemporary MDE development environment.

4. Evaluate and assess the proposed structures and processes for managing evolutionary change, particularly with respect to the productivity and understandability of software development.

1.5 Research Method

To explore the hypothesis outlined above, the thesis research was conducted using the method described in this section and summarised in Figure 1.1. The green boxes represent the three *phases* of research, which are described below. The white boxes represent inputs and outputs to those phases.

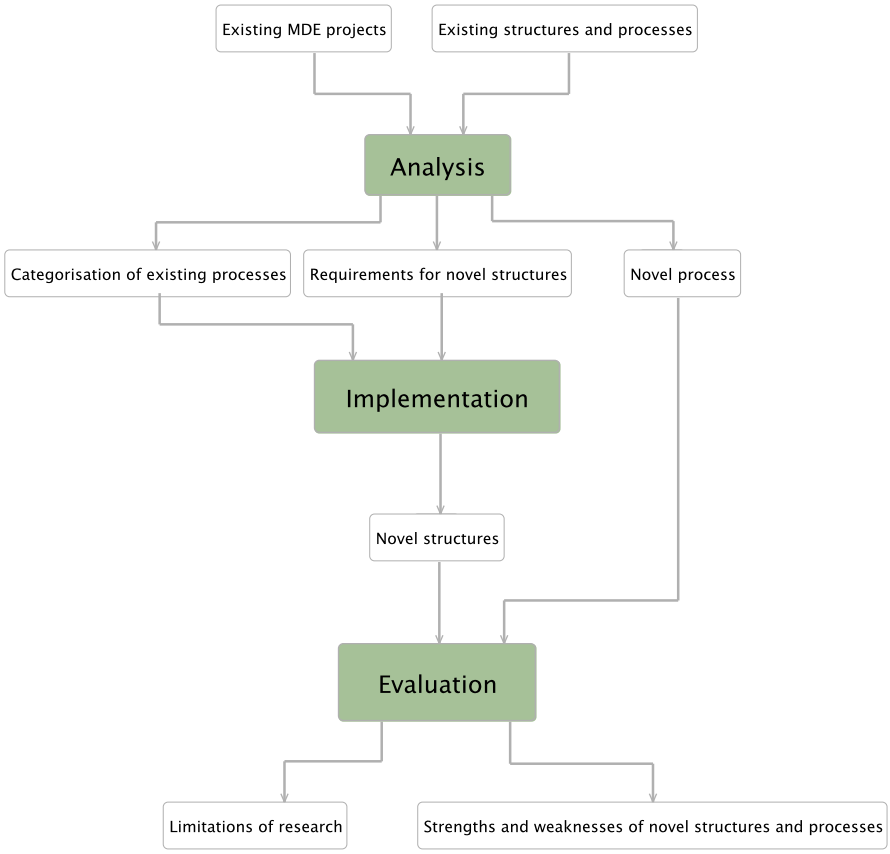


Figure 1.1: Overview of the research method.

Firstly, the *analysis* phase involved studying the evolution of MDE development artefacts in existing projects. The results of the analysis phase were used to determine a category of evolution that lacked support in contemporary MDE development environments, *model-metamodel co-evolution* or, simply *co-evolution*. Co-evolution examples from existing MDE projects were used to categorise existing processes for managing co-evolution, and to formulate requirements for new structures and processes for managing co-evolution. The analysis phase also lead to the identification of *user-driven co-evolution*, a process for managing co-evolution that has not previously been recognised

in the co-evolution literature.

The *implementation* phase involved proposing, designing and implementing novel structures for managing co-evolution, and integrating the structures with a contemporary MDE environment. The co-evolution examples identified in the analysis phase were used for testing the implementation of the structures.

The *evaluation* phase involved assessing the novel structures for managing co-evolution by comparison to existing structures, and demonstrating the novel process. Evaluation was performed using further examples of co-evolution. To mitigate a possible threat to the validity of the research, the examples used in the evaluation phase were different to those used in the analysis phase. The strengths and weaknesses of the novel and existing structures and processes were synthesised from the comparisons, particularly with respect to the productivity and understandability of software development.

A similar method was used successfully in [Dig 2007] to explore the extent to which component-based applications can be automatically evolved. Initially, [Dig & Johnson 2006b] conducted *analysis* to identify and categorise evolution in five existing component-based applications, with the hypothesis that many of the changes could be classified as behaviour-preserving. By using examples from the survey, [Dig *et al.* 2006] were able to *implement* an algorithm for automatically detecting behaviour-preserving changes. The algorithm was then used to implement tools for (1) migrating code in a distributed and collaborative software development environment [Dig & Johnson 2006a], and (2) analysing the history of component-based applications [Dig *et al.* 2007]. The latter facilitated better understanding of program evolution, and refinement of the detection algorithm. Finally, [Dig 2007] *evaluated* the tools and detection algorithm by application to three further component-based applications.

1.6 Research Results

This thesis proposes novel structures and processes for managing model-metamodel co-evolution. Reference implementations of the proposed structures have been constructed, including *Epsilon HUTN* (a textual modelling notation) and *Epsilon Flock* (a model migration language). The reference implementations have been constructed atop Epsilon [Kolovos 2009], an extensible platform for specifying MDE languages and tools, and are interoperable with the Eclipse Modelling Framework [Steinberg *et al.* 2008], arguably the most widely used MDE modelling framework.

Additionally, this thesis proposes a novel process for managing model-metamodel co-evolution and proposes a theoretical categorisation of existing process for managing model-metamodel co-evolution. The novel process, termed *user-driven co-evolution*, is demonstrated by application to a MDE

development process for a real-world project.

The research hypothesis has been validated by comparing the proposed structures and processes with existing structures and processes using examples of evolution from real-world MDE projects. Evaluation has been performed using several approaches, including a collaborative comparison of model migration tools carried out with three MDE experts, comparing quantitive measurements of the proposed and existing migration languages, and application of the proposed structures and processes to two examples of evolution, including an example from a widely used modelling language, the Unified Modelling Language (UML) [OMG 2007a].

1.7 Thesis Structure

Chapter 2 gives an overview of MDE by defining terminology; describing associated engineering principles, practices and tools; and reviewing related areas of computer science. Section 2.5 synthesises some of the benefits of, and challenges for, contemporary MDE.

Chapter 3 reviews theoretical and practical software evolution research. Areas of research that underpin software evolution are described, including refactoring, design patterns, and traceability. The review then discusses work that approaches particular categories of evolution problem, such as programming language, schema and grammar evolution. Section 3.2.4 surveys work that considers evolution in the context of MDE. Section 3.3 identifies three types of evolution that occur in MDE projects and highlights challenges for their management.

Chapter 4 surveys existing MDE projects and categories the evolution of MDE development artefacts in those projects. From this survey, the context for the thesis research is narrowed, and the remainder of the thesis focuses on one type of evolution occurring in MDE projects, termed *model-metamodel co-evolution* or simply *co-evolution*. Examples of co-evolution are used to identify the strengths and weaknesses of existing structures and processes for managing co-evolution. From this, Section 4.2.2 identifies a process for managing co-evolution which has not been recognised previously in the literature, Section 4.2.3 derives a categorisation of existing processes for managing co-evolution, and Section 4.3 synthesis requirements for novel structures for managing co-evolution.

Chapter 5 describes novel structures for managing co-evolution, including a metamodel-independent syntax, which is used to identify, report and to facilitate the reconciliation of problems caused by metamodel evolution. The textual modelling notation described in Section 5.2 and the model migration language described in Section 5.4 are used for reconciliation of models in response to metamodel evolution. The latter provides a means for performing reconciliation in a repeatable manner.

Chapter 6 assesses the structures and processes proposed in this thesis by comparison to existing structures and processes. To explore the research hypothesis, several different types of comparison were performed, including an experiment in which quantitive measurements were derived, a collaborative comparison of model migration tools with three MDE experts, and application to a large, independent example of evolution taken from a real-world MDE project.

Chapter 7 summarises the achievements of the research, and discusses results in the context of the research hypothesis. Limitations of the thesis research and areas of future work are also outlined.

Chapter 2

Background

This chapter surveys literature from the area in which the thesis research was conducted, Model-Driven Engineering (MDE). MDE is a principled approach to software engineering in which models are produced and consumed throughout the engineering process. Section 2.1 introduces the terminology and fundamental principles used in MDE. Section 2.2 reviews guidance and three methods for performing MDE. Section 2.3 describes contemporary MDE environments. Two areas of research relating to MDE, domain-specific languages and language-oriented programming, are discussed in Section 2.4. Finally, the benefits of and current challenges for MDE are described in Section 2.5.

2.1 MDE Terminology and Principles

Software engineers using MDE construct and manipulate artefacts familiar from traditional approaches to software engineering (such as code and documentation) and, in addition, work with different types of artefact, such as *models*, *metamodels* and *model transformations*. Furthermore, MDE involves new development activities, such as *model management*. This section describes the artefacts and activities involved in MDE.

2.1.1 Models

Models are fundamental to MDE. [**?**] identifies many definitions of the term model, such as: "any subject using a system A that is neither directly nor indirectly interacting with a system B to obtain information about the system B, is using A as a model for B." [**?**], "a model is a representation of a concept. The representation is purposeful and used to abstract from reality the irrelevant details." [**?**], and "a model is a simplification of a system written in a well-defined language." [**?**].

While there are many definitions of the term model, a common notion is that a model is a representation of the real-world [**?**, pg12]. The part of the

real-world represented by a model is termed the *domain*, the *object system* or, simply the *system*. A further commonality is noted by [Kolovos *et al.* 2006]: a model may have either a textual or graphical representation.

[**?**] defines *analogous* models as those which share some characteristics and can be used in place of their object system. An aeroplane toy that can fly is an analogous model of an aeroplane. In computer science, models can be used to construct a computer system. A model of an object system, say the lending service of a library, might be used to decide the way in which data is stored on disk, or the way in which a program is to be structured.

[**?**] proposes that the models constructed in computer science are analogous to two systems: the object system (e.g. the library lending service in the real-world) and the computer system (e.g. the combination of software and hardware used to implement a library lending service). A model can be used to think about both the real system and the computer system. Figure 2.1 illustrates this notion further. According to [**?**], a model is both the description of the domain (object system) and the machine (computer system). Computer scientists switch between *designations* when using a model to think about the object system or to think about the software system.

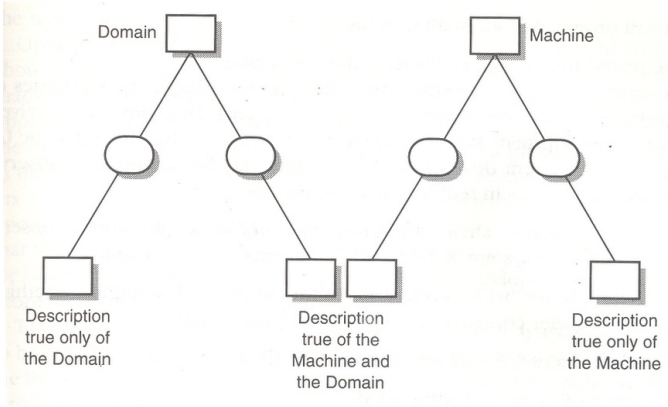


Figure 2.1: Jackson's definition of a model, taken from [**?**, pg.125].

Models can be unstructured (for example, sketches on a piece of paper) or structured (conform to some well-defined set of syntactic and semantic constraints). In software engineering, models are used widely to reason about object systems and computer systems. MDE recognises this, and seeks to drive the development of computer systems from structured models.

2.1.2 Modelling languages

In MDE, models are structured (satisfy a well-defined set of syntactic and semantic constraints) rather than unstructured [Kolovos 2009]. A *modelling*

language is the set of syntactic and semantic constraints used to define the structure of a group of related models. In MDE, a modelling language is often specified as a model and, hence the term *metamodel* is used in place of *modelling language*.

Conformance is a relationship between a metamodel and a model. A model *conforms to* a metamodel when the metamodel specifies every concept used in the model definition, and the model uses the metamodel concepts according to the rules specified by the metamodel. Conformance can be described by a set of constraints between models and metamodels [**?**]. When all constraints are satisfied, a model conforms to a metamodel. For example, a conformance constraint might state that every object in the model has a corresponding non-abstract class in the metamodel.

Metamodels facilitate model interchange and, therefore, interoperability between modelling tools. For this reason, Evans recommends that software engineers "use a well-documented shared language that can express the necessary domain information as a common medium of communication." [**?**, pg377]. To support this recommendation, Evans discusses the way in which chemists have collaborated to define a standardised language for describing chemical structures, Chemical Markup Language (CML)[1]. The standardisation of CML has facilitated interoperability between tools for specification, analysis and simulation.

A metamodel typically comprises three categories of constraint:

- **The concrete syntax** provides a notation for constructing models that conform to the language. For example, a model may be represented as a collection of boxes connected by lines. A standardised concrete syntax enables communication. Concrete syntax may be optimised for consumption by machines (e.g. XML Metadata Interchange (XMI) [OMG 2007c]) or by humans (e.g. the concrete syntax of the Unified Modelling Language (UML) [OMG 2007a]).

- **The abstract syntax** defines the concepts described by the language, such as classes, packages, datatypes. The representation for these concepts is independent of the concrete syntax. For example, the implementation of a compiler might use an abstract syntax tree to encode the abstract syntax of a program (whereas the concrete syntax for the same language may be textual or graphical).

- **The semantics** identifies the meaning of the modelling concepts in the particular domain of language. For example, consider a modelling language defined to describe genealogy, and another to describe flora. Although both languages may define a tree construct, the semantics of a tree in one is likely to be different from the semantics of a tree

[1] `http://cml.sourceforge.net/`

in the other. The semantics of a modelling language may be specified rigorously, by defining a reference semantics in a formal language such as Z [**?**], or in a semi-formal manner by employing natural language.

Concrete syntax, abstract syntax and semantics are used together to specify modelling languages. There are many other ways of defining languages, but this approach (first formalised in [**?**]) is common in model-driven engineering: a metamodel is often used to define abstract syntax, a grammar or text-to-model transformation to specify concrete syntax, and code generators, annotated grammars or behavioural models to effect semantics.

2.1.3 MOF: A metamodelling language

Software engineers using MDE can use existing and define new metamodels. To facilitate interoperability between MDE tools, the OMG has standardised a language for specifying metamodels, the Meta-Object Facility (MOF). Metamodels specified in MOF can be interchanged between MDE environments. Furthermore, modelling language tools are interoperable because MOF also standardises the way in which metamodels and their models are persisted to and from disk. For model and metamodel persistence, MOF prescribes XML Metadata Interchange (XMI), a dialect of XML optimised and standardised by the OMG for loading, storing and exchanging models.

Because MOF is a modelling language for describing modelling languages, it is sometimes termed a metamodelling language. Part of the UML metamodel, defined in MOF, is shown in Figure 2.2. As discussed in Section 2.3, different kinds of concrete syntax can be used for MOF. Figure 2.2, for example, uses a concrete syntax similar to that of UML class diagrams. Specifically:

- Modelling constructs are drawn as boxes. The name of each modelling construct is emboldened. The name of abstract (uninstantiable) constructs are italicised.
- Attributes are contained within the box of their modelling construct. Each attribute has a name, a type (prefixed with a colon) and may define a default value (prefixed with an equals sign).
- Generalisation is represented using a line with an open arrow-head.
- References are specified using a line. An arrow illustrates the direction in which the reference may be traversed (no arrow indicates bi-directionality). Labels are used to name and define the multiplicity of references.
- Containment references are specified by including a solid diamond on the containing end.

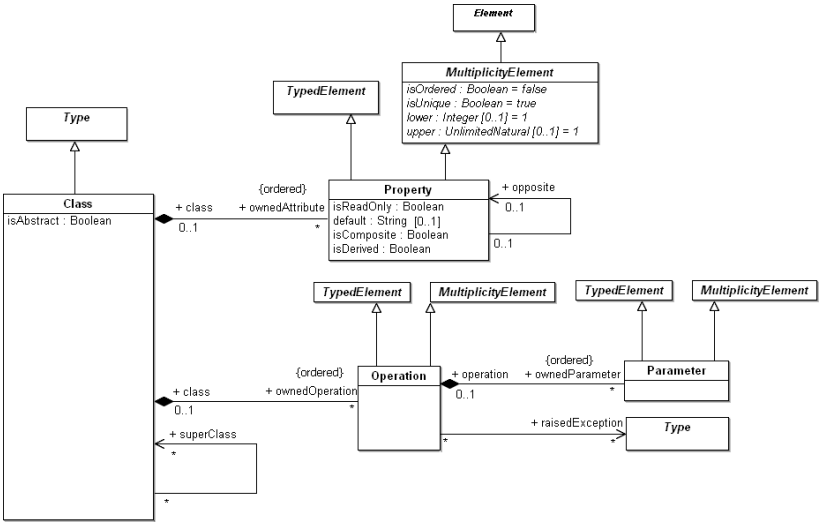


Figure 2.2: A fragment of the UML metamodel defined in MOF, from [OMG 2007a].

Specifying modelling languages with a common metamodelling language, such as MOF, ensures consistency in the way in which modelling constructs are specified. MOF has facilitated the construction of interoperable MDE tools that can be used with a range of modelling languages. Without a standardised metamodelling language, modelling tools were specific to one modelling language, such as UML. In contemporary MDE environments, any number of modelling languages can be used together and manipulated in a uniform manner.

Furthermore, when modelling languages are specified without using a common metamodelling language, identifying similarities between modelling languages is challenging [Frankel 2002, pg97]. The sequel discusses the way in which models and metamodels are used to construct systems in MDE.

2.1.4 Model Management

In MDE, models are *managed* to produce software. [**?**] first described *model management* as a collection of operators for manipulating models. [Kolovos 2009] explores a means for increasing the interoperability of model management operations. This thesis uses the term *model management* to refer to development activities that manipulate models for the purpose of producing software. Model management activities typical in MDE, such as model transformation and validation, are discussed in this section. Section 2.2 discusses MDE guidelines and methods, and describes the way in which model management activities are used together to produce software in MDE.

Model Transformation

Model transformation is a development activity in which software artefacts are derived from others, according to some well-defined specification. Three different types of model transformation are described in [Kleppe *et al.* 2003, Kolovos 2009]. Model transformations are specified between modelling languages (model-to-model transformation), between modelling languages and textual artefacts (model-to-text-transformation) and between textual artefacts and modelling languages (text-to-model transformation). Each type of transformation has unique characteristics and tools, but share some common characteristics. The remainder of this section first introduces the commonalities and then discusses each type of transformation individually.

Common characteristics of model transformations The input to a transformation is termed its *source*, and the output its *target*. In theory, a transformation can have more than one source and more than one target, but not all transformation languages support multiple sources and targets. Consequently, much of the model transformation literature considers single source and target transformations.

[Czarnecki & Helsen 2006] describes a feature model for distinguishing and categorising model transformation approaches. Two of the features are relevant to the research presented in this thesis, and are now discussed.

Source-target relationship A *new-target* transformation creates target models afresh on each invocation. An *existing-target* transformation is executed on existing target models. Existing target transformations are used for partial (incremental) transformation and for preserving parts of the target that are not derived from the source.

Domain language Transformations specified between a source and a target model that conform to the same metamodel are termed *endogenous* or *rephrasings*, while transformations specified between a source and a target model that conform to different metamodels are termed *exogenous* or *translations*.

Endogenous, existing-target transformations are a special case of transformation and are termed *refactorings*. Refactorings have been studied in the context of software evolution and are discussed more thoroughly in Chapter 3.

Model-to-Model (M2M) Transformation M2M transformation is used to derive models from others. By automating the derivation of models from others, M2M transformation has the potential to reduce the cost of engineering large and complex systems that can be represented as a set of interdependent models [**?**].

M2M transformations are often specified as a set of *rules* [Czarnecki & Helsen 2006]. Each rule specifies the way in which a specific set of elements in the source model is transformed to an equivalent set of elements in the target model [Kolovos 2009, pg.44].

Many M2M transformation languages have been proposed, such as the Atlas Transformation Language (ATL) [Jouault & Kurtev 2005], the Epsilon Transformation Language (ETL) [Kolovos *et al.* 2008a] and VIATRA [**?**]. The OMG [OMG 2008b] provide a standardised M2M transformation language, Queries/Views/Transformations (QVT) [OMG 2005]. M2M transformation languages can be categorised according to their *style*, which is either declarative, imperative or hybrid.

Declarative M2M transformation languages only provide constructs for mapping source to target model elements and, as such, are not computationally complete. Consequently, the scheduling of rules can be *implicit* (determined by the execution engine of the transformation language). By contrast, imperative M2M transformation languages are computationally complete, but often require rule scheduling to be *explicit* (specified by the user). Hybrid M2M transformation languages combine declarative and imperative parts, are computationally complete, and provide a mixture of implicit and explicit rule scheduling.

Because declarative M2M transformation languages cannot be used to solve some categories of transformation problem [**?**] and imperative M2M transformation languages are argued to be difficult to write and maintain [Kolovos 2009, pg.45], [Kolovos *et al.* 2008a] notes that the current consensus is that hybrid languages, such as ATL are more suitable for specifying model transformation than pure imperative or declarative languages.

An exemplar M2M transformation, written in the hybrid M2M transformation language ETL, is shown in Listing 2.1. The source of the transformation is a state machine model, conforming to the metamodel shown in Figure 2.3. The target of the transformation an object-oriented model, conforming to the metamodel shown in Figure 2.4. The transformation in Listing 2.1 comprises two rules.

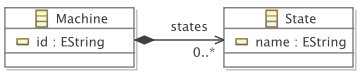


Figure 2.3: Exemplar State Machine metamodel.

```
1  rule Machine2Package
2    transform m : StateMachine!Machine
3    to      p : ObjectOriented!Package {
4
```

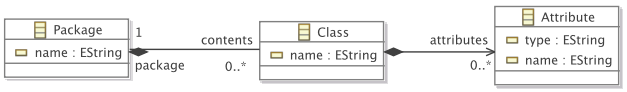


Figure 2.4: Exemplar Object-Oriented metamodel.

```
5   p.name  := 'uk.ac.york.cs.' + m.id;
6   p.contents := m.states.equivalent();
7  }
8
9  rule State2Class
10   transform s : StateMachine!State
11   to      c : ObjectOriented!Class
12
13   guard: not s.isFinal {
14
15   c.name := s.name + 'State';
16  }
```

Listing 2.1: Exemplar M2M transformation in the Epsilon Transformation Language [Kolovos *et al.* 2008a]

The first rule (lines 1-7) is named `Machine2Package` (line 1) and transforms *Machine*s (line 2) into *Package*s (line 3). The body of the first rule (lines 5-6) specifies the way in which a `Package`, `p`, can be derived from a `Machine`, `m`. Specifically, the `name` of `p` is derived from the `id` of `m` (line 5), and the `contents` of `p` are derived from the `states` of `m` (line 6).

The second rule (lines 9-16) transforms `States` (line 10) to `Classes` (line 11). Additionally, line 13 contains a *guard* to specify that the rule is only to be applied to `States` whose `isFinal` property is `false`.

When executed, the transformation rules will be scheduled **implicitly** by the execution engine, and invoked once for each `Machine` and `State` in the source. On line 6 of Listing 2.1, the built-in `equivalent()` operation is used to produce a set of `Classes` from a set of `States` by invoking the relevant transformation rule. This is an example of **explicit** rule scheduling, in which the user defines when a rule will be called.

Model-to-Text (M2T) Transformation M2T transformation is used for model serialisation (enabling model interchange), code and documentation generation, and model visualisation and exploration. In 2005, the OMG [OMG 2008b] recognised the lack of a standardised M2T transformation with its M2T Language Request for Proposals [2]. In response, various M2T lan-

[2] `http://www.omg.org/docs/ad/04-04-07.pdf`

guages have been developed, including JET[3], XPand[4], MOFScript [Oldevik *et al.* 2005] and the Epsilon Generation Language (EGL) [Rose *et al.* 2008].

Because M2T transformation is used to produce unstructured rather than structured artefacts, M2T transformation has different requirements to M2M transformation. For instance, M2T transformation languages often provide mechanisms for specifying sections of text that will be completed manually and must not be overwritten by the transformation engine.

*Template*s are commonly used in M2T languages. Templates comprise *static* and *dynamic* sections. When the transformation is invoked, the contents of static sections are emitted verbatim, while dynamic sections contain logic and are executed.

An exemplar M2T transformation, written in EGL, is shown in Listing 2.2. The source of the transformation is an object-oriented model conforming to the metamodel shown in Figure 2.4, and the target is Java source code. The template assumes that an instance of `Class` is stored in the `class` variable.

```
1  package [%=class.package.name];
2
3  public class [%=class.name] {
4    [% for(attribute in class.attributes) { %]
5     private [%=attribute.type%] [%=attribute.name];
6    [% } %]
7  }
```

Listing 2.2: Exemplar M2T transformation in the Epsilon Generation Language [Rose *et al.* 2008]

In EGL, dynamic sections are contained within `[%` and `%]`. *Dynamic output* sections are a specialisation of dynamic sections contained within `[%=` and `%]`. The result of evaluating a dynamic output section is included in the generated text. Line 1 of Listing 2.2 contains two static sections (`package'` and `;`) and a dynamic output section (`[%=class.package.name]`), and will generate a package declaration when executed. Similarly, line 3 will generate a class declaration. Lines 4 to 6 iterate over every `attribute` of the `class`, outputting a field declaration for each `attribute`.

Text-to-Model (T2M) Transformation T2M transformation is most often implemented as a parser that generates a model rather than object code. Parser generators such as ANTLR [Parr 2007] can be used to produce a structured artefact (such as an abstract syntax tree) from text. T2M tools are built atop parser generators and post-process the structured artefacts such that they conform to a metamodel specified by the user.

Xtext[5] and EMFtext [**?**] are contemporary examples of T2M tools that,

[3] `http://www.eclipse.org/modeling/m2t/?project=jet#jet`
[4] `http://www.eclipse.org/modeling/m2t/?project=xpand`
[5] `http://www.eclipse.org/Xtext/`

given a grammar and a target metamodel, will automatically generate a parser that transforms text to a model.

An exemplar T2M transformation, written in EMFtext, is shown in Listing 2.3. From the transformation shown in Listing 2.3, EMFtext can be used to generate a parser that, when executed, will produce state machine models. For the input, `lift[stationary up down stopping emergency]`, the parser will produce a model containing one `Machine` with `lift` as its `id`, and five `States` with the `names`, `stationary`, `up`, `down`, `stopping`, and `emergency`.

```
1  SYNTAXDEF statemachine
2  FOR <statemachine>
3  START Machine
4
5  TOKENS {
6    DEFINE IDENTIFIER $('a'..'z'|'A'..'Z')*$;
7    DEFINE LBRACKET $'['$;
8    DEFINE RBRACKET $']'$;
9  }
10
11 RULES {
12   Machine ::= id[IDENTIFIER] LBRACKET states* RBRACKET ;
13   State ::= name[IDENTIFIER] ;
14 }
```

Listing 2.3: Exemplar T2M transformation in EMFtext

Lines 1-2 of Listing 2.3 define the name of the parser and target metamodel. Line 3 indicates that parser should first seek to construct a `Machine` from the source text. Lines 5-9 define rules for the lexer, including a rule for recognising `IDENTIFIERs` (represented as alphabetic characters).

Lines 11-14 of Listing 2.3 are key to the transformation. Line 11 specifies that a `Machine` is constructed whenever an `IDENTIFIER` is followed by a `LBRACKET` and eventually a `RBRACKET`. When constructing a `Machine`, the first time an `IDENTIFIER` is encountered, it is stored in the `id` attribute of the `Machine`. The `states*` statement on line 12 indicates that, before matching a `RBRACKET`, the parser is permitted to transform subsequent text to a `State` (according to the rule on line 13) and store the resulting `State` in the `states` reference of the `Machine`. The asterisks in `states*` indicates that any number of `States` can be constructed and stored in the `states` reference.

Model Validation

Model validation provides a mechanism for managing the integrity of the software developed using MDE. A model that omits information is said to be *incomplete*, while related models that suggest differences in the underlying phenomena are said to be *contradicting* [Kolovos 2009]. Incompleteness

and contradiction are two examples of *inconsistency*. In MDE, inconsistency is detrimental, because, when artefacts are automatically derived from each other, the inconsistency of one artefact might be propagated to others. Model validation is used to detect, report and reconcile inconsistency throughout a MDE process.

[Kolovos 2009] observes that inconsistency detection is inherently pattern-based and, hence, higher-order languages are more suitable for model validation than so-called "third-generation" programming languages (such as Java). The Object Constraint Language (OCL) [OMG 2006] is an OMG standard that can be used to specify consistency constraints on UML and MOF models. OCL cannot be used to specify inter-model constraints, unlike the xlinkit toolkit [**?**] and the Epsilon Validation Language (EVL) [Kolovos *et al.* 2008b].

An exemplar model validation constraint, written in EVL, is shown in Listing 2.4. The constraint validates state machine models that conform to the metamodel shown in Figure 2.3. The constraint shown in Listing 2.4 is defined for `States` (line 1), and checks that there exists some transition whose source or target is the current state (line 4). When the check part (line 4) is not satisfied, the message part (line 6) is displayed. When executed, the EVL constraint will be invoked once for every `State` in the model. The keyword `self` is used to refer to the particular `State` on which the constraint is currently being invoked.

```
1  context State {
2    constraint NoStateIsAnIsland {
3      check:
4        Transition.all.exists(t | t.source == self or t.target == self)
5      message:
6        'The state ' + self.name + ' has no transitions.'
7    }
8  }
```

Listing 2.4: Exemplar model validation in the Epsilon Validation Language

Further model management activities

In addition to model transformation and validation, further examples of model management activities include model comparison (e.g. [**?**]), in which a *trace* of similar and different elements is produced from two or more models, and model merging or weaving (e.g. [**?**]), in which two or more models are combined to produce a unified model.

Further activities, such as model versioning and tracing, might be regarded as model management but, in the context of this thesis, are considered as evolutionary activities and as such are discussed in Chapter 3.

2.1.5 Summary

This section has introduced the terminology and principles necessary for discussing MDE in this thesis. Models provide abstraction, capturing necessary and disregarding irrelevant details. Metamodels provide a structured mechanism for describing the syntactic and semantic rules to which a model most conform. Metamodels facilitate interoperability between modelling tools and MOF, the OMG standard metamodelling language, enables the development of tools that can be used with a range of metamodels, such as model management tools. Throughout model-driven engineering, models are manipulated to produce other development artefacts using model management activities such as model transformation and validation. Using the terms and principles described in this section, the ways in which model-driven engineering is performed in practice are now discussed.

2.2 MDE Guidelines and Methods

For performing MDE, new engineering practices and processes have been proposed. Proponents of MDE have produced guidance and methods for MDE. This section discusses the guidance for MDE set out in the Model-Driven Architecture [**?**] and the methods for MDE described in [**?**, **?**, **?**].

2.2.1 Model-Driven Architecture (MDA)

Model-Driven Architecture (MDA) is a software engineering framework defined by the OMG. MDA provides guidelines for MDE. For instance, MDA prescribes the use of a Platform Independent Model (PIM) and one or more Platform Specific Models (PSMs).

A PIM provides an abstract, implementation-agnostic view of the solution. Successive PSMs provide increasingly more implementation detail. Inter-model mappings are used to forward- and reverse-engineer these models, as depicted in Figure 2.5.

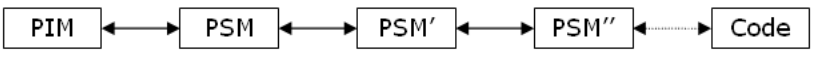


Figure 2.5: Interactions between a PIM and several PSMs.

A key difference between MDA and related approaches, such as round-trip engineering (in which models and code are co-evolved to develop a system), is that MDA prescribes automated transformations between PIM and PSMs, whereas other approaches use some manual transformations.

Standards for MDA

As part of the guidelines for MDE, the MDA prescribes a set of standards. The standards are allocated to one of four tiers, and each tier represents a different levels of model abstraction. Members of one tier conform to a member of the tier above. The four tiers described in the MDA are shown in Figure 2.6, and a short discussion based on [Kleppe *et al.* 2003, Section 8.2] follows.

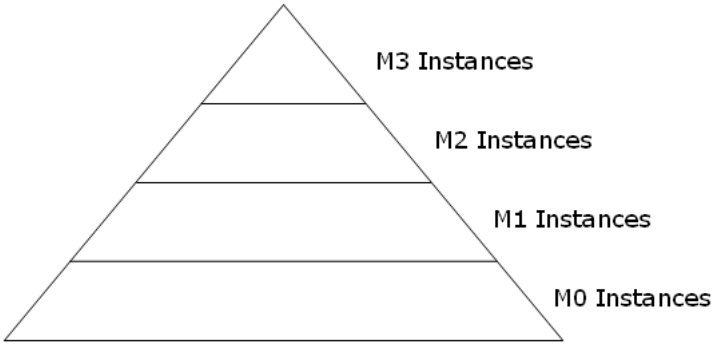


Figure 2.6: The tiers of standards used as part of MDA.

The base of the pyramid, tier M0, contains the domain (real-world). When modelling a business, this tier is used to describe items of the business itself, such as a real customer or an invoice. When modelling software, M0 instances describe the software representation of such items. M1 contains models (Section 2.1.1) of the concepts in M0, for example a customer may be represented as a class with attributes. The M2 tier contains the modelling languages (metamodels, Section 2.1.2) used to describe the contents of the M1 tier. For example, if UML [OMG 2007a] models were used to describe concepts as classes in the M1 tier, M2 would contain the UML metamodel. Finally, M3 contains a metamodelling language (metametamodel, Section 2.1.3) which describes the modelling languages in the M2 tier. As discussed in Section 2.1.3, the M3 tier facilitates tool standardisation and interoperability. The MDA specifies the Meta-Object Facility (MOF) [OMG 2008a] as the only member of the M3 tier.

Interpretations of MDA

[**?**] identifies two ways in which engineers have interpreted MDA. Both interpretations begin with a PIM, but the way in which executable code is produced varies:

- **Translationist**: The PIM is used to generate code directly using a sophisticated code generator. Any intermediate PSMs are internal to the code generator. No generated artefacts are edited manually.

- **Elaborationist**: Any generated artefacts (such as PSMs, code and documentation) can be augmented with further details of the application. To ensure that all models and code are synchronised, tools must allow bi-directional transformations.

Translationists must encode behaviour in their PIMs [**?**], whereas elaborationists have a choice, frequently electing to specify behaviour in PSMs or in code [Kleppe *et al.* 2003].

The difference between translationist and elaborationist approaches to MDE is related to a difference in the way in which models are viewed in traditional approaches to software engineering. For example, [**?**] proposes the use of models throughout the development process, and the way in which code is structured is driven by the model. By contrast, [**?**, ch14] prescribes modelling only for communicating and reasoning about a design, and not "as a long-term replacement for real, working software". Rather [**?**] advocates using models to quickly compare different ways in which a system might be structured and then to disregard those models in favour of working code.

2.2.2 Methods for MDE

Several methods for MDE are prevalent today. In this section, three of the most established MDE methods are discussed: Architecture-Centric Model-Driven Software Development [**?**], Domain-Specific Modelling [**?**] and Microsoft's Software Factories [**?**]. All three methods have been defined by MDE practitioners, and have been used repeatedly to solve problems in industry. The methods vary in the extent to which they follow the guidelines set out by MDA.

Architecture-Centric Model-Driven Software Development

Model-Driven Software Development is the term given to MDE by in [**?**]. The style of MDE that [**?**] describes, *architecture-centric model-driven software development* (AC-MDSD), focuses on generating the infrastructure of large-scale applications. For example, a typical J2EE application contains concepts (such as EJBs, descriptors, home and remote interfaces) that "admittedly contain domain-related information such as method signatures, but which also exhibit a high degree of redundancy" [**?**]. It is this redundancy that AC-MDSD seeks to remove by using code generators, requiring only the domain-related information to be specified.

AC-MDSD applies more of the MDA guidelines than the other methods discussed below. For instance, AC-MDSD supports the use of a general-purpose modelling language for specifying models. [**?**] utilise UML in many of their examples, which demonstrate how AC-MDSD may be used to enhance the productivity, efficiency and understandability of software development. In

these examples, models are annotated using UML profiles to describe domain-specific concepts.

Domain-Specific Modelling

[?] present Domain-Specific Modelling (DSM), a collection of principles, practices and advice for constructing systems using MDE. DSM is based on the translationist interpretation of MDA: domain models are transformed directly to code. In motivating the need for DSM, Kelly and Tolvanen state that large productivity gains were made when third-generation programming languages were used in place of assembler, and that no paradigm shift has since been able to replicate this degree of improvement. Tolvanen[6] notes that DSM focuses on increasing the productivity of software engineering by allowing developers to specify solutions by using models that describe the application domain.

To perform DSM, expert developers define:

- **A domain-specific modelling language**: allowing domain experts to encode solutions to their problems.
- **A code generator**: that translates the domain-specific models to executable code in an existing programming language.
- **Framework code**: that encapsulates the common areas of all applications in this domain.

As the development of these three artefacts requires significant effort from expert developers, Tolvanen[6] states that DSM should only be applied if more than three problems specific to the same domain are to be solved.

Tools for defining domain-specific modelling languages, editors and code generators enable DSM [?]. Reducing the effort required to specify these artefacts is key to the success of DSM. In this respect, DSM resembles a programming paradigm termed *language-oriented programming* (LOP), which also requires tools to simplify the specification of new languages. LOP is discussed further in Section 2.4.

Throughout [?], examples from industrial partners are used to argue that DSM can improve developer productivity. Unlike MDA, DSM appears to be optimised for increasing productivity, and less concerned with portability or maintainability. Therefore, DSM is less suitable for engineering applications that frequently interoperate with – and are underpinned by – changing technologies.

[6]Tutorial on Domain Specific Modelling for Full Code Generation at the Fourth European Conference on Model Driven Architecture (ECMDA), June 2008, Berlin, Germany.

Microsoft Software Factories

[**?**, pg159] states that industrialisation of the automobile industry has addressed problems with economies of scale (mass production) and scope (product variation). Software Factories, a software engineering method developed at Microsoft, seeks to address problems with economies of scope in software engineering by borrowing concepts from product-line engineering. [**?**] argues that, unlike many other engineering disciplines, software development requires considerably more development effort than production effort in that scaling software development to account for scope is significantly more complicated then mass production of the same software system.

The Software Factories method [**?**] prescribes a bottom-up approach to abstraction and re-use. Development begins by producing prototypical applications. The common elements of these applications are identified and abstracted into a product-line. When instantiating a product, models are used to choose values for the variation points in the product. To simplify the creation of these models, Software Factories propose model creation wizards. Greenfield et al. state that "moving from totally-open ended hand-coding to more constrained forms of specification [such as wizard-based feature selection] are the key to accelerating software development" [**?**, pg179]. By providing explanations that assist in making decisions, the wizards used in Software Factories guide users towards best practices for customising a product.

Compared to DSM, the Software Factories method appears to provide more support for addressing portability problems. The latter provides *viewpoints* into the product-line (essentially different ways of presenting and aggregating data from development artefacts), which allow decoupling of concerns (e.g. between logical, conceptual and physical layers). Viewpoints provide a mechanism for abstracting over different layers of platform independence, adhering more closely than DSM to the guidelines provided in MDA. Unlike the guidelines provided in MDA, the Software Factories method does not insist that development artefacts be derived automatically where possible.

Finally, the Software Factories method prescribes the use of domain-specific languages (discussed in Section 2.4.1) for describing models in conjunction with Software Factories, rather than general-purpose modelling languages, as the authors of Software Factories believe that the latter often have imprecise semantics [**?**].

2.2.3 Summary

This section has discussed the ways in which process and practices for MDE have been captured. Guidance for MDE has been set out in the MDA standard, which seeks to use MDE to produce adaptable software in a productive and maintainable manner. Three methods for performing MDE have been discussed.

The methods discussed share some characteristics. They all require a set of exemplar applications, which are examined by MDE experts. Analysis of the exemplar applications identifies the way in which software development may be decomposed. A modelling language for the problem domain is constructed, and instances are used to generate future applications. Code common to all applications in the problem domain is encapsulated in a framework.

Each method has a different focus. AC-MDSD seeks to automatically generate code that repeats information from the problem domain, particularly for enterprise applications. The Software Factories method concentrates on providing different viewpoints into the system, and facilitating collaborative specification of a system. DSM aims to improve reusability between solutions to problems in the same problem domain, and hence improve developer productivity.

Perhaps unsurprisingly, the proponents of each method for MDE recommend a single tool (such as MetaCase for DSM). Alternative tools are available from open-source modelling communities, including the Eclipse Modelling Project, which provides – among other MDE tools – arguably the most widely used MDE modelling framework today. Two MDE tools are reviewed in the sequel.

2.3 MDE Tools

For MDE to be applicable in the large, and to complex systems, mature and powerful tools and languages must be available. Such tools and languages are beginning to emerge This section discusses two MDE tools that are well-suited for MDE research and are used in the remainder of the thesis. Although other MDE tools exist, there are not used for the thesis research and not reviewed in this section.

Section 2.3.1 provides an overview of the Eclipse Modelling Framework (EMF) [**?**], which implements MOF and underpins many contemporary MDE tools and languages, facilitating their interoperability. Section 2.3.2 discusses Epsilon [**?**], an extensible platform for the specification of model management languages. The highly extensible nature of Epsilon (which is described below) makes it an ideal host for the rapid prototyping of languages and exploring research hypotheses.

The purpose of this section is to review EMF and Epsilon, which are used throughout the remainder of the thesis, and not to provide a thorough review of all MDE tools. There are many other MDE tools and environments that this section does not discuss, such as ATL [ATLAS 2007] and VIATRA [**?**] for M2M transformation, oAW [openArchitectureWare 2007] for model transformation and validation, MOFScript [Oldevik *et al.* 2005] and XPand [**?**] for M2T transformation, and the AMMA [**?**] platform for large-scale modelling, model weaving and software modernisation.

2.3.1 Eclipse Modelling Framework (EMF)

[?] is an open-source community seeking to build an extensible development platform. The Eclipse Modelling Framework (EMF) project [?] enables MDE within Eclipse. EMF provides a modelling framework with code generation facilities, and a meta-modelling language, Ecore, that implements the MOF 2.0 specification [OMG 2008a]. EMF is arguably the most widely-used contemporary MDE modelling framework.

EMF is used to generate metamodel-specific editors for loading, storing and constructing models. EMF model editors comprise a navigation view for specifying the elements of the model, and a properties view for specifying the features of model elements. Figure 2.7 shows an EMF model editor for a simplistic state machine language. The navigation (or tree) view is shown in the top pane, while the properties view is shown in the bottom pane.

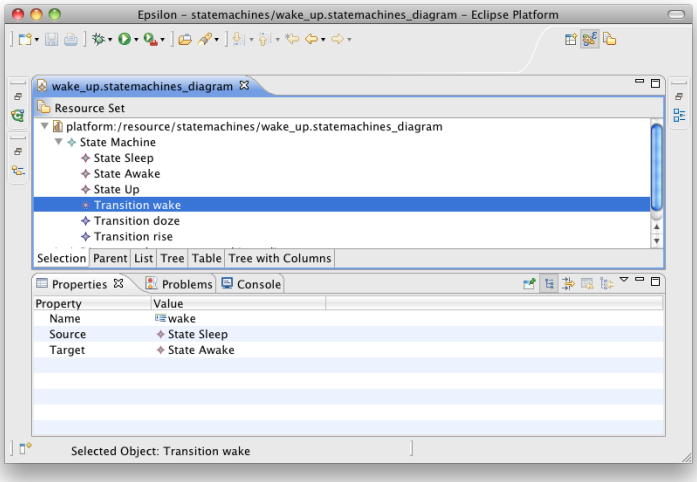


Figure 2.7: An EMF model editor for state machines.

Users of EMF can define their own metamodels in Ecore, the metamodelling language and MOF implementation of EMF. EMF provides two metamodel editors, tree-based and graphical. Figure 2.8 shows the metamodel of a simplistic state machine language in the tree-based metamodel editor. Figure 2.9 shows the same metamodel in the graphical metamodel editor. Like MOF, the graphical metamodel editor uses concrete syntax similar to that of UML class diagrams. Emfatic [IBM 2005] provides a further, textual metamodel editor for EMF, and is shown in Figure 2.10. The editors shown in Figure 2.8, 2.9 and 2.10 are being used to manipulate the same underlying metamodel, but using different syntaxes. A change to the metamodel in one editor can be propagated automatically to the other two.

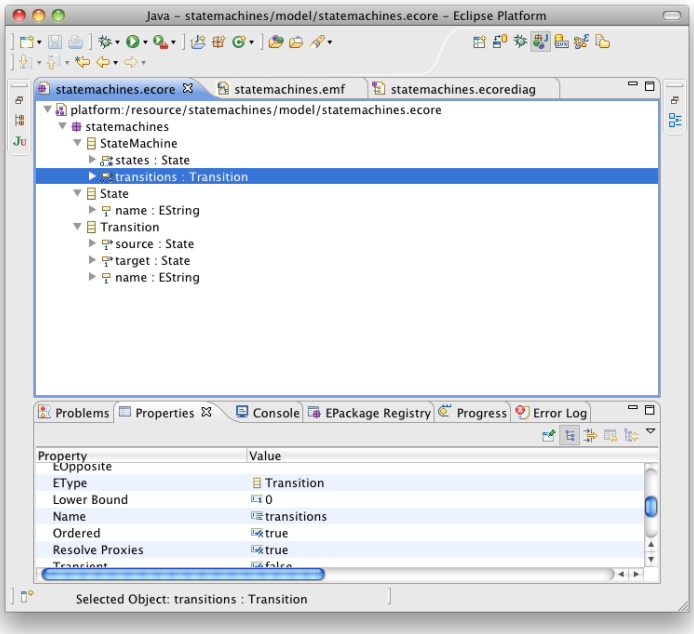


Figure 2.8: EMF's tree-based metamodel editor.

From a metamodel, EMF can generate an editor for models that conform to that metamodel. For example, the simplistic state machine metamodel specified in Figures 2.8, 2.9 and 2.10 was used to generate the code for the model editor being used in Figure 2.7. The model editors generated by EMF include mechanisms for persisting models to and from disk. As prescribed by MOF, EMF typically generates code that persists models using XMI [OMG 2007c], a dialect of XML optimised for model interchange.

The Graphical Modeling Framework (GMF) [Gronback 2009] is used to specify generate graphical model editors from metamodels defined with EMF. Figure 2.11 shows a model editor produced with GMF for the simplistic state machine language described above. GMF itself uses a model-driven approach: users specify several models, which are combined, transformed and then used to generate code for the resulting graphical editor.

Many MDE tools are interoperable with EMF, enriching its functionality. The remainder of this section discusses one tool that is interoperable with EMF, Epsilon, which is a suitable platform for rapid prototyping of model management languages and, hence, is useful for performing MDE research.

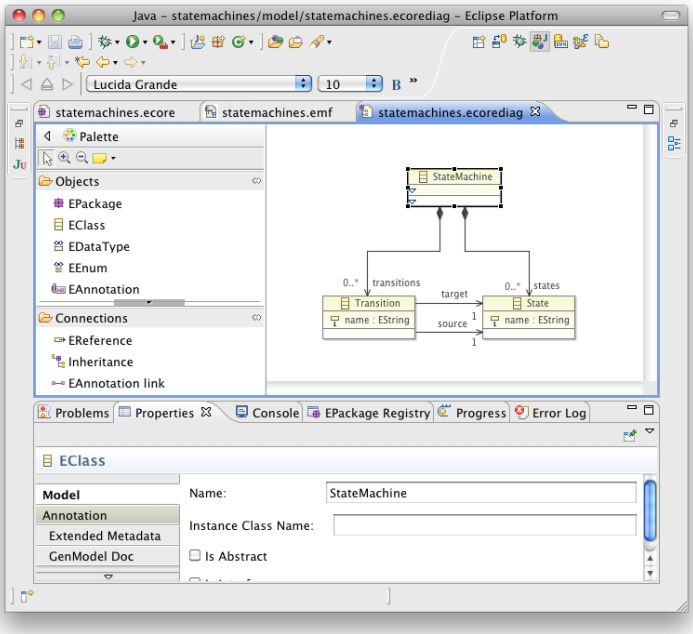


Figure 2.9: EMF's graphical metamodel editor.

2.3.2 Epsilon

The Extensible Platform for Specification of Integrated Languages for mOdel maNagement (Epsilon) [**?**] is a suite of tools and domain-specific languages for MDE. Epsilon comprises several integrated model management languages – built atop a common infrastructure – for performing tasks such as model merging, model transformation and inter-model consistency checking [Kolovos 2009]. Figure 2.12 illustrates the various components of Epsilon.

Whilst many model management languages are bound to a particular subset of modelling technologies, limiting their applicability, Epsilon is metamodel-agnostic: models written in any modelling language can be manipulated by Epsilon's model management languages [Kolovos *et al.* 2006]. Currently, Epsilon supports models implemented using EMF, MOF 1.4, XML, or Community Z Tools (CZT)[7]. Interoperability with further modelling technologies can be achieved by extension of the Epsilon Model Connectivity (EMC) layer.

The architecture of Epsilon promotes reuse when building task-specific model management languages and tools. Each Epsilon language can be reused wholesale in the production of new languages. Ideally, the developer of a new language only has to design language concepts and logic that do not

[7] `http://czt.sourceforge.net/`

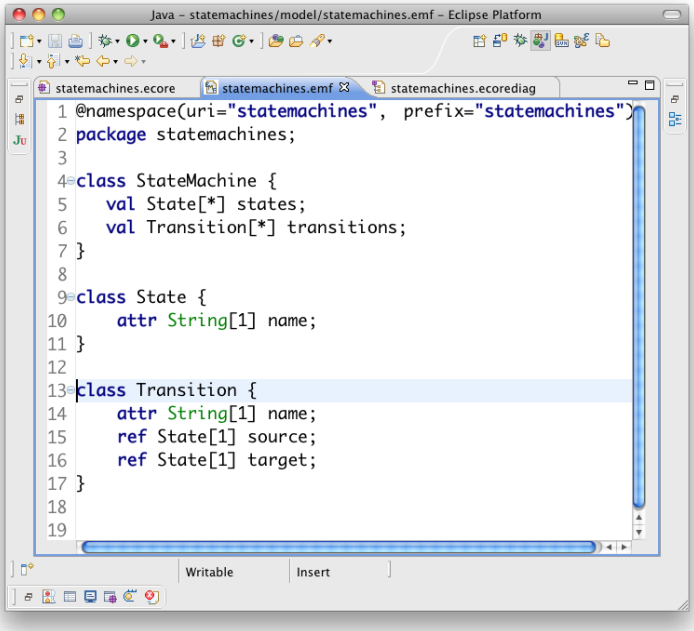


Figure 2.10: The Emfatic textual metamodel editor for EMF.

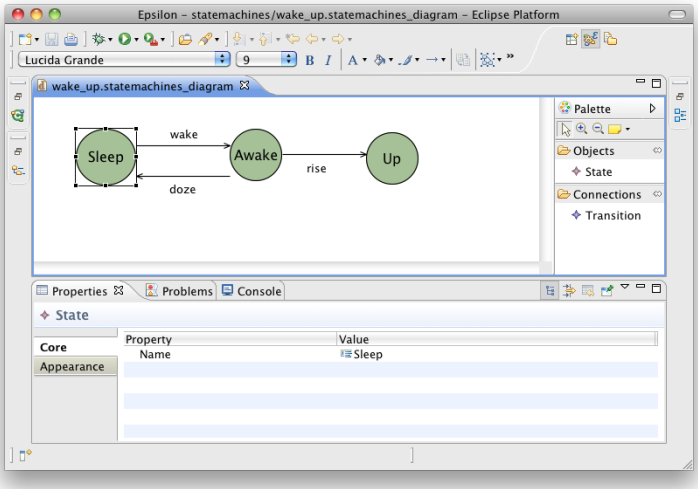


Figure 2.11: GMF state machine model editor.

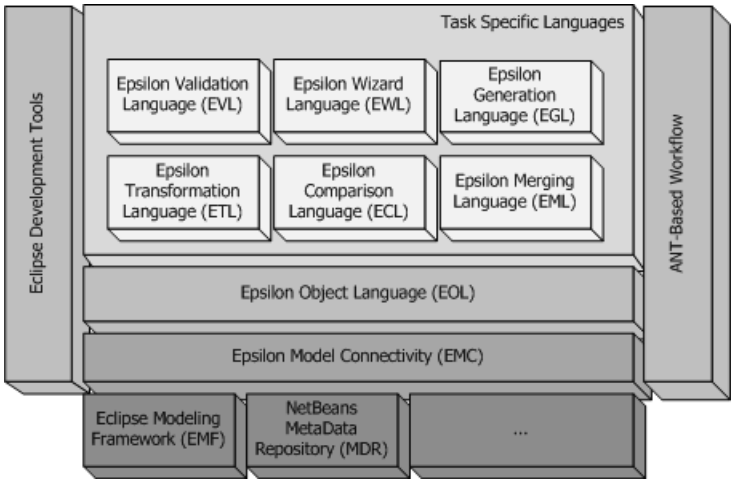


Figure 2.12: The architecture of Epsilon, taken from [Rose *et al.* 2008].

already exist in Epsilon languages. As such, new task-specific languages can be implemented in a minimalistic fashion. This claim has been demonstrated in [Rose *et al.* 2008], which describes the Epsilon Generation Language (EGL) for specifying M2T transformation. Epsilon has been used extensively for the work described in Chapter 5.

The Epsilon Object Language (EOL) [Kolovos *et al.* 2006] is the core of the platform and provides functionality similar to that of OCL [OMG 2006]. However, EOL provides an extended feature set, which includes the ability to update models, access to multiple models, conditional and loop statements, statement sequencing, and provision of standard output and error streams.

As shown in Figure 2.12, every Epsilon language re-uses EOL, so improvements to this language enhance the entire platform. EOL also allows developers to delegate computationally intensive tasks to extension points, where the task can be authored in Java.

Epsilon is a member of the Eclipse GMT [Eclipse 2008] project, a research incubator for the top-level modelling technology project. Epsilon provides a lightweight means for defining new experimental languages for MDE. For these reasons, Epsilon is uniquely positioned as an ideal host for the rapid prototyping of languages for model management.

2.3.3 Summary

This section has introduced the MDE tools used throughout the remainder of the thesis. The Eclipse Modeling Framework (EMF) provides an implementation of MOF, Ecore, for defining metamodels. From metamodels defined in Ecore, EMF can generate code for metamodel-specific editors and for per-

sisting models to disk. EMF is arguably the most widely used contemporary MDE modelling framework and its functionality is enhanced by numerous tools, such as the Graphical Modeling Framework (GMF) and Epsilon. GMF allows metamodel developers to specify a graphical concrete syntax for metamodels, and can be used to generate graphical model editors. Epsilon is an extensible platform for defining and executing model management languages, provides a high degree of re-use for defining new model management languages and can be used with a range of modelling frameworks, including EMF.

2.4 Research Relating to MDE

MDE is closely related to several other engineering and software development fields. This section discusses two of those fields, Domain-Specific Languages (DSLs) and Language-Oriented Programming (LOP). A further related area, Grammarware, is discussed in the context of software evolution in Section 3.2.3. DSLs and LOP are closely related to the research central to this thesis. Other areas relating to MDE but less relevant to this thesis, such as formal methods, are not considered here.

2.4.1 Domain-Specific Languages

For a set of closely-related problems, a specific, tailored approach is likely to provide better results than instantiating a generic approach for each problem [**?**]. The set of problems for which the specific approach outperforms the generic approach is termed the *domain*. A *domain-specific programming language* (often called a *domain-specific language* (DSL)) enables the encoding of solutions for a particular domain.

Like modelling languages, DSLs describe abstract syntax. Furthermore, a common language can be used to define DSLs (e.g. EBNF [**?**]), like the use of MOF for defining modelling languages. In addition to abstract syntax, DSLs typically define a textual concrete syntax but, like modelling languages, can utilise a graphical concrete syntax.

Cobol, Fortran and Lisp first existed as DSLs for solving problems in the domains of business processing, numeric computation and symbolic processing respectively, and evolved to become general-purpose programming languages [**?**]. SQL, on the other hand, is an example of a DSL that, despite undergoing much change, has not grown into a general-purpose language. Unlike a general-purpose language, a single DSL cannot be used to program an entire application. DSLs are often small languages at inception, but can grow to become complicated (such as SQL). Within their domain, DSLs should be easy to read, understand and edit [**?**].

There are two ways in which DSLs are typically implemented. An *internal* DSL is implemented by describing the domain using constructs from a general-purpose language (the *host*) [**?**, **?**]. Examples of internal DSLs include

the frameworks for working with collections that are included in some programming languages (e.g. STL for C++, the Collections API for Java). Some languages are better than others for hosting internal DSLs. For example, [**?**] proposes Ruby as a suitable host for DSLs due to its "unintrusive syntax and flexible runtime evaluation." [**?**] describes a technique for implementing internal DSLs in Lisp, in which macros are used to translate domain-specific concepts to Lisp abstractions.

[**?**] reports that internal DSLs can exhibit some unsatisfactory characteristics because there is often a mismatch between domain and programming abstractions. For this reason, [**?**] prefers to implement DSLs by *translating* DSL programs into code written in a general-purpose language. [**?**] uses the term *external* for this style of DSL implementation. Programs written in simple DSLs are often easy to translate to programs in an existing general-purpose language [Parr 2007]. Approaches to translation include preprocessing; building or generating an interpreter or compiler; or extending an existing compiler or interpreter [**?**].

The construction of an external DSL can be achieved using many of the principles, practices and tools used in MDE. Parsers can be generated using text-to-model transformation; syntactic constraints can be specified with model validation; and translation can be specified using model-to-model and model-to-text transformation. MDE tools are used to implement two external DSLs in Chapter 5.

Internal and external DSLs have been successfully used as part of application development in many domains, as described in [**?**]. They have been used in conjunction with general-purpose languages to build systems rapidly and to improve productivity in the development process (such as automation of system deployment and configuration). More recently, some developers are building complete applications by combining DSLs, in a style of development called Language-Oriented Programming.

2.4.2 Language-Oriented Programming

[**?**] coins the term Language-Oriented Programming (LOP) to describe a style of development in which a very high-level language is used to encode the problem domain. Simultaneously, a compiler is developed to translate programs written in the high-level language to an existing programming language. Ward describes how this approach to programming can enhance the productivity of development and the understandability of a system. Additionally, Ward mentions the way in which multiple very high-level languages could be layered to separate domains.

The high-level languages that Ward discusses are domain-specific. [**?**] notes that combining DSLs to solve a problem is not a new technique. Traditionally, UNIX has encouraged developers to combine programs written in small (domain-specific) languages (such as awk, make, sed, lex and yac) to solve

problems. Lisp, Smalltalk and Ruby programmers often construct domain-specific languages when developing programs [**?**, **?**].

To fully realise the benefits of LOP, the development effort required to construct DSLs must be minimised. Two approaches for constructing DSLs seem to be prevalent for LOP. The first advocates using a highly dynamic, reflexive and extensible programming language to specify DSLs. [**?**] terms this category of language a *superlanguage*. The superlanguage permits new DSLs to re-use constructs from existing DSLs, which simplifies development.

A *language workbench* [**?**] is an alternative means for simplifying DSL development. Language workbenches provide tools, wizards and DSLs for defining abstract and concrete syntax, for constructing editors and for specifying code generators.

For defining DSLs, the main difference between using a language workbench or a superlanguage is the way in which semantics of language concepts are encoded. In a language workbench, a typical approach is to write a generator for each DSL (e.g. MPS [**?**]), whereas a superlanguage often requires that semantics be encoded in the definition of language constructs (e.g. XMF [**?**]).

Like MDE, LOP requires mature and powerful tools and languages to be applicable in the large, and to complex systems. Unlike MDE, LOP tools typically combine concrete and abstract syntax. The emphasis for LOP is in defining a single, textual concrete syntax for a language. MDE tools might provide more than one concrete syntax for a single modelling language. For example, two distinct concrete syntaxes are used for the tree-based and graphical editors of the simplistic state-machine language shown in Figures 2.7 and 2.11.

Some of the key concerns for MDE are also important to the success of LOP. For example, tools for performing LOP and MDE need to be as usable as those available for traditional development, which often include support for code-completion, automated refactoring and debugging. Presently, these features are often lacking in tools that support LOP or MDE.

In summary, LOP addresses many of the same issues with traditional development as MDE, but requires a different style of tool. LOP focuses more on the integration of distinct DSLs, and providing editors and code generators for them. Compared to LOP, MDE typically provides more separation between concrete and abstract syntax, and concentrates more on model management.

2.4.3 Summary

This section has described two areas of research related to MDE, domain-specific languages (DSLs) and language-oriented programming (LOP). DSLs facilitate the encoding of solutions for a particular problem domain. For solving problems in their domain, DSLs can be easier to read, use and edit than general-purpose programming languages [**?**, **?**]. During MDE, one or more DSLs may be used to model the domain, and the tools and techniques for

implementing DSLs can be used for MDE.

LOP is an approach to software development that seeks to specify complete systems using a combination of DSLs. Contemporary LOP seeks to minimise the effort required to specify and use DSLs. Like MDE, LOP requires mature and powerful tools, but, unlike MDE, LOP does not separate concrete and abstract syntax, and does not focus on model management, which is a key development activity in MDE.

2.5 Benefits of and Current Challenges for MDE

Compared to traditional software engineering approaches and to domain-specific languages and language-oriented programming, MDE has several benefits and weaknesses. This section identifies benefits of and challenges to MDE, synthesised from the literature reviewed in this chapter.

2.5.1 Benefits

Three benefits of MDE are now identified, and used to describe the advantages of the MDE principles and practices discussed in this chapter.

Tool interoperability MOF, the standard metamodelling language for MDE, facilitates interoperability between tools via model interchange. With Ecore, EMF provides a reference implementation of MOF atop which many contemporary MDE tools are built. Interoperability between modelling tools allows model management to be performed across a range of tools, and developers are not tied to one vendor. Furthermore, models represented in a range of modelling languages can be used together in a single environment. Prior to the formulation of MOF, developers would use different tools for each modelling language. Each tool would likely have different storage formats, complicating the interchange of models between tools.

Managing complexity For software systems that must incorporate large-scale complexity, such as those that support large businesses, managing stochastic interaction in the large is a key concern. With MDE it is possible to sacrifice total reliability or validity of a system to achieve a working solution. Sacrificing reliability or validity is not always possible when other engineering approaches are used to construct software (such as formal methods).

System evolution The guidelines set out for MDE in MDA [**?**] highlight principles and patterns for modelling to increase the adaptability of software systems by, for example, separating platform-specific and platform-independent detail. When the target platform changes (for example a new technological

architecture is required), only part of the system needs to be changed. The platform-independent detail can be re-used wholesale.

Related to this, MDE facilitates automation of the error-prone or tedious elements of software engineering. For example, code generation can be used to automatically produce so-called "boilerplate" code, which is repetitive code that cannot be restructured to remove duplication (typically for technological reasons).

While MDE can be used to reduce the extent to which a system is changed in some circumstances, MDE also introduces additional challenges for managing system evolution [Mens & Demeyer 2007]. For example, mixing generated and hand-written code typically requires a more elaborate software architecture than would be used for a system composed of only hand-written code. Further examples of the challenges that MDE presents for evolution are discussed in the sequel.

2.5.2 Challenges

Three challenges for MDE are now identified, and used to motivate areas of potential research for improving MDE. The remainder of the thesis focuses on the final challenge, maintainability in the small.

Learnability MDE involves new terminology, development activities and principles for software engineering. For the novice, producing a simple system with MDE is arguably challenging. For example, [Kolovos *et al.* 2009] explores the steps required to generate a graphical model editor with the Graphical Modeling Framework (GMF), concludes that GMF is difficult for new users to understand, and presents a mechanism for simplifying GMF for new users. It seems reasonable to assume that the extent to which MDE tools and principles can be learnt will eventually determine the adoption rate of MDE.

Scalability As discussed in [Rose *et al.* 2010b], in traditional approaches to software engineering a model is considered of comparable value to any other documentation artefact, such as a word processor document or a spreadsheet. As a result, the convenience of maintaining self-contained model files which can be easily shared outweighs other desirable attributes. [Kolovos *et al.* 2008c] notes that this perception has led to the situation where single-file models of the order of tens (if not hundreds) of megabytes, containing hundreds of thousands of model elements, are the norm for real-world software projects.

MDE languages and tools must scale such that they can be used with with large and complex models. [Hearnden *et al.* 2006, Ráth *et al.* 2008, Tratt 2008] explore ways in which the scalability of model management tasks, such as model transformation, can be improved. [Kolovos *et al.* 2008c] prescribes a different approach, suggesting that MDE research should aim for greater modularity in models, which, as a by-product, will result in greater scalability in

MDE. Scalability of MDE tools is a key concern for practitioners and, for this reason, [Kolovos *et al.* 2008c] terms scalability the "holy grail" of MDE.

Development artefact evolution Notwithstanding the benefits of MDE for managing the evolution of systems, the introduction of additional development artefacts (such as models and metamodels) and activities (such as model management) presents additional challenges for the way in which developers manage software evolution [Mens & Demeyer 2007]. For example, in traditional approaches to software engineering, maintainability is primarily achieved by restructuring code, updating documentation and regression testing [**?**]. It is not yet clear the extent to which existing maintenance activities can be applied in MDE. (For example, should models be tested and, if so, how?)

As demonstrated in Chapter 4, the way in which some MDE tools are structured limits the extent to which some traditional maintenance activities can be performed. Understanding, improving and assessing the way in which evolution is managed in the context of MDE is an open research topic to which this thesis contributes.

2.5.3 Summary

This section has identified some of the benefits of and challenges for contemporary MDE. The interoperability of tools and modelling languages in MDE allows developers greater flexibility in their choice of tools and facilitates interchange between heterogenous tools and modelling frameworks. MDE is more flexible than other, more formal approaches to software engineering, which can be beneficial for constructing complex systems. The principles and practices of MDE can be used to achieve greater maintainability of systems by, for example, separating platform-independent and platform-specific details.

As MDE tools approach maturity, non-functional requirements, such as learnability, and scalability, become increasingly desirable for practitioners. MDE tools must also be able to support developers in managing changing software. This section has demonstrated some of the weakness of contemporary MDE, particularly in the areas of learnability, scalability and supporting software evolution.

2.6 Chapter Summary

To be completed.

Chapter 3

Literature Review

This chapter provides a review and critical analysis of existing work on software evolution and identifies potential research directions. The principles of software evolution are discussed in Section 3.1, while Section 3.2 reviews the ways in which evolution is identified, analysed and managed in a range of fields, including relational databases, programming languages, and model-driven development environments. From the reviewed literature, Section 3.3 synthesises research challenges for software evolution in the context of MDE, highlighting those to which this thesis contributes, and elaborates on the research method used in this thesis.

3.1 Software Evolution Theory

Software evolution is an important facet of software engineering. Studies [Erlikh 2000, Moad 1990] suggest that the evolution of software can account for as much as 90% of a development budget. Such figures are sometimes described as uncertain [Sommerville 2006, ch. 21], primarily because the term evolution is not used consistently. Nonetheless, there is a corpus of software evolution research, and publications in this area have existed since the 1960s (e.g. [Lehman 1969]).

The remainder of this section introduces software evolution terminology and discusses three research areas that relate to software evolution: refactoring, design patterns and traceability. Refactoring concentrates on improving the structure of existing systems, design patterns on best practices for software design, and traceability for recording and analysing the lifecycle of software artefacts. Each area provides a common vocabulary for discussing software design and evolution. There is an abundance of research in these areas, including seminal works on refactoring by [Opdyke 1992] and [Fowler 1999]; and on design patterns by [Alexander *et al.* 1977] and [Gamma *et al.* 1995].

3.1.1 Categories of Software Evolution

[Sjøberg 1993] identifies reasons for software evolution, which include addressing changing requirements, adapting to new technologies, and architectural restructuring. These reasons are the motivations for three common types of software evolution [Sommerville 2006, ch. 21]:

- **Corrective evolution** takes place when a system exhibiting unintended or faulty behaviour is corrected. Alternatively, corrective evolution may be used to adapt a system to new or changing requirements.
- **Adaptive evolution** is employed to make a system compatible with a change to platforms or technologies that underpin its implementation.
- **Perfective evolution** refers to the process of improving the internal quality of a system, while preserving the behaviour of the system.

The remainder of this section adopts this categorisation for discussing software evolution literature. Refactoring (discussed in Section 3.1.2), for instance, is one way in which perfective evolution can be realised.

Many activities are used for managing software evolution. [Winkler & Pilgrim 2009] highlight the importance of *impact analysis* (for reasoning about the effects of evolution) and *change propagation* (for updating one artefact in response to a change made to another). In addition, [Sommerville 2006] notes that *reverse engineering* (analysing existing development artefacts to extract information) and *source code translation* (rewriting code to use a more suitable technology, such as a different programming language) are also important software evolution activities. MDE facilitates portable software, for example by prescribing platform-independent and platform-specific models (as discussed in Section 2.1.4), and as such source code translation is arguably less relevant to MDE than to traditional software engineering. Because MDE seeks to capture the essence of the software in models, reverse engineering information from, for example, code is also less likely to be relevant to MDE than to tradition software engineering. Consequently, this thesis focuses on impact analysis and change propagation.

3.1.2 Refactoring

[Mens & Tourwé 2004] report "an urgent need for techniques that reduce software complexity by incrementally improving the internal software quality." Refactoring was first described by [Opdyke 1992] and is "the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure" [Fowler 1999, pg. xvi]. Refactoring plays a significant role in the evolution of software systems – a recent study of five open-source projects showed that over 80% of changes were refactorings [Dig & Johnson 2006b].

Typically, refactoring literature concentrates on three primary activities in the refactoring process: *identification* (where should refactoring be applied, and which refactorings should be used?), *verification* (has refactoring preserved behaviour?) and *assessment* (how has refactoring affected other qualities of the system, such as cohesion and efficiency?).

In the foreword to [Fowler 1999], Beck describes an informal means for identifying the need for refactoring, termed *bad smells*: "structures in the code that suggest (sometimes scream for) the possibility of refactoring.". Tools and semi-automated approaches have also been devised for refactoring identification, such as Daikon [Kataoka *et al.* 2001], which detects program invariants that may indicate the possibility for refactoring. Clone analysis tools have been employed for identifying refactorings that eliminate duplication [Balazinska *et al.* 2000, Ducasse *et al.* 1999]. The types of refactoring being performed may vary over different domains. For example, Buck[1] describes refactorings, such as "Skinny Controller, Fat Model", particular to the Ruby on Rails web framework [37-Signals 2008].

MOF [OMG 2008a], discussed in Section 2.1.4, provides a standard notation for describing the abstract syntax of metamodels. As MOF re-uses many concepts from UML class diagrams (which are used to describe the structure of object-oriented systems), object-oriented refactorings can be applied to metamodels defined using MOF. However, no standard means has yet been defined for attaching semantics to modelling language constructs. When a metamodel is defined without a rigorous semantics, refactoring of the sort applied to OO code does not seem to be directly applicable. (In particular, drawing parallels to existing approaches for the verification and assessment activities of refactoring seems difficult). Regardless, refactoring catalogues, such as [Fowler 1999], might influence the way in which model evolution is recorded, due to the clarity and conciseness of their format. This is discussed further in Section 3.1.3.

Since 2006, Dig has been studying the refactoring of systems that are developed by combining components, possibly developed by different organisations. [Dig & Johnson 2006b] reports a survey used to identify and categorise the changes made to five components that are known to have been re-used often, with the hypothesis that a significant number of the changes could be classified as behaviour-preserving (i.e. refactorings). By using examples from the survey, [Dig *et al.* 2006] devises an algorithm for automatically detecting refactorings to a high degree of accuracy (over 85%). The algorithm was then utilised in tools for (1) replaying refactorings to perform migration of client code following breaking changes to a component [Dig & Johnson 2006a], and (2) versioning object-oriented programs using a refactoring-aware configuration management system [Dig *et al.* 2007]. The latter facilitated better under-

[1] In a keynote address to the First International Ruby on Rails Conference (RailsConf), May 2007, Portland, Oregon, United States of America.

standing of program evolution, and the refinement of the refactoring detection algorithm.

3.1.3 Patterns and anti-patterns

A *design pattern* identifies a commonly occurring design problem and describes a re-usable solution to that problem. Related design patterns are combined to form a *pattern catalogue* – such as for object-oriented programming [Gamma *et al.* 1995] or enterprise applications [Fowler 2002]. A pattern description comprises at least a name, overview of the problem, and details of a common solution [Brown *et al.* 1998]. Depending on the domain, further information may be included in the pattern description (such as a classification, a description of the pattern's applicability and an example usage).

Design patterns can be thought of as describing objectives for improving the internal quality of a system (perfective software evolution). [Kerievsky 2004] provides a practical guide that describes how software can be refactored towards design patterns to improve its quality. Studying the way in which experts perform perfective software evolution can lead to devising best practices, sometimes in the form of a pattern catalogue, such as the object-oriented refactorings described in [Fowler 1999].

[Alexander *et al.* 1977] first used design patterns when devising a pattern catalogue for town planning. [Beck & Cunningham 1989] later adapted the work of Alexander for software architecture, by specifying a pattern catalogue for designing user-interfaces. Utilising pattern catalogues allowed the software industry to "reuse the expertise of experienced developers to repeatedly train the less experienced." [Brown *et al.* 1998, pg. 10]. [Rising 2001, pg. xii] summarises the usefulness of design patterns: "Patterns help to define a vocabulary for talking about software development and integration challenges; and provide a process for the orderly resolution of these challenges."

Anti-patterns are an alternative literary form for describing patterns of a software architecture [Brown *et al.* 1998]. Rather than describe patterns that have often been observed in successful architectures, they describe those which are present in unsuccessful architectures. Essentially, an anti-pattern is a pattern in an inappropriate context, which describes a problematic solution to a frequently encountered problem. The (anti-)pattern catalogue may include alternative solutions that are known to yield better results (termed "refactored solutions" by [Brown *et al.* 1998]). Catalogues might also consider the reasons why (inexperienced) developers might select an anti-pattern. Brown notes that "patterns and anti-patterns are complementary" [Brown *et al.* 1998, pg. 13]; both are useful in providing a common vocabulary for discussion of system architectures and in educating less experienced developers.

3.1.4 Traceability

A software development artefact rarely evolves in isolation. Changes to one artefact cause and are caused by changes to other artefacts (e.g. object code is recompiled when source code changes, source code and documentation are updated when requirements change). Hence, traceability – the ability to describe and follow the life of software artefacts [Winkler & Pilgrim 2009, Lago *et al.* 2009] – is closely related to and facilitates software evolution.

Historically, traceability is a branch of requirements engineering, but increasingly traceability is used for artefacts other than requirements [Winkler & Pilgrim 2009]. Because MDE prescribes automated transformation between models, traceability is also researched in the context of MDE. The remainder of this section discusses traceability principles focussing on the relationship between traceability and software evolution, while Section 3.2.4 reviews the traceability literature that relates to MDE.

Traceability is facilitated by *traceability links*, which document the dependencies, causalities and influences between artefacts. Traceability links are established by hand or by automated analysis of artefacts. In MDE environments, some traceability links can be automatically inferred because the relationships between some types of artefact are specified in a structured manner (for example, as a model-to-model transformation).

Traceability links are defined between artefacts at the same level of abstraction (horizontal links) and at different levels of abstraction (vertical links). Uni-directional traceability links are navigated either *forwards* (away from the dependent artefact) or *backwards* (toward the dependent artefact). Figure 3.1 summaries these categories of traceability link.

The traceability literature uses inconsistent terminology. This thesis adopts the same terminology as [Winkler & Pilgrim 2009]: *traceability* is the ability to describe and follow the life of software artefacts; *traceability links* are the relationships between software artefacts.

Traceability supports software evolution activities, such as impact analysis (discovering and reasoning about the effects of a change) and change propagation (updating impacted artefacts following a change to an artefact). Moreover, automated software evolution is facilitated by programmatic access to traceability links.

Current approaches for traceability-supported software evolution use *triggers* and *events*. Each approach proposes mechanisms for detecting triggers (changes to artefacts) and for notifying dependent artefacts of events (the details of a change). Existing approaches vary in the extent to which they can automatically update dependent artefacts. The approaches described in [Chen & Chou 1999, Cleland-Huang *et al.* 2003] report inconsistencies and do not perform automatic updates, while [Aizenbud-Reshef *et al.* 2005, Costa & Silva 2007] propose reactive approaches for guided or fully automatic updates. Section 3.2.4 provides a more thorough discussion and critical analysis of event-

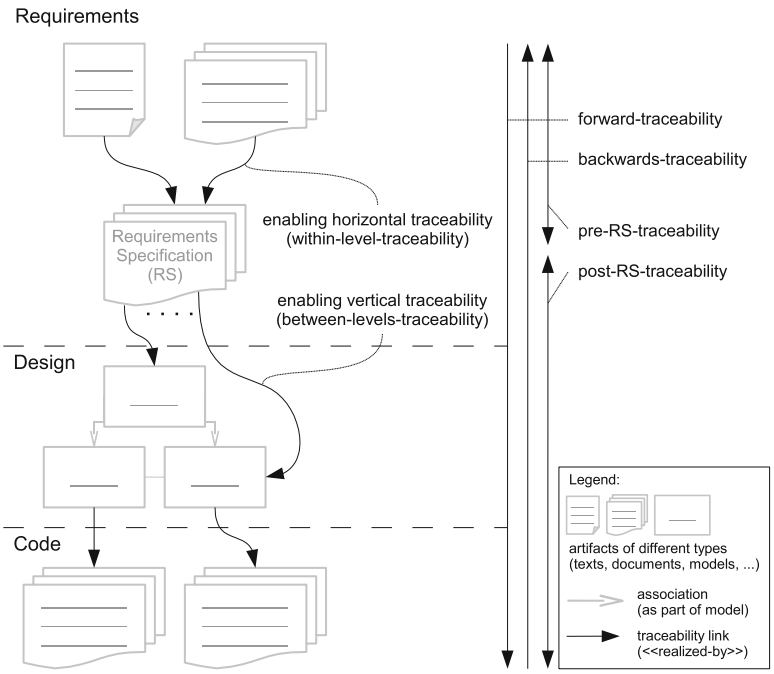


Figure 3.1: Categories of traceability link [Winkler & Pilgrim 2009].

based approaches for impact analysis and change propagation in the context of MDE.

To remain accurate and hence useful, traceability links must be updated as a system evolves. Although most existing approaches to traceability are "not well suited to the evolution of [traceability] artefacts" [Winkler & Pilgrim 2009, pg. 24], there is some work in this area. For example, [Mäder *et al.* 2008] describe a development environment that records changes to artefacts, comparing the changes to a catalogue of built-in patterns. Each pattern provides an executable specification for updating traceability links.

Software evolution and traceability are entangled concerns. Traceability facilitates software evolution activities such as impact analysis and change propagation. Traceability is made possible with consistent and accurate traceability links. Software evolution can affect the relationships between artefacts (i.e. the traceability links) and hence software evolution techniques are applied to ensure that traceability links remain consistent and accurate.

3.2 Software Evolution in Practice

Using the principles of software evolution described above, this section examines the ways in which evolution is identified, managed and analysed in a variety of settings, including programming languages grammarware, relational database management system and MDE.

3.2.1 Programming Language Evolution

Programming language designers often attempt to ensure that legacy programs continue to conform to new language specifications. For example, [Cervelle *et al.* 2006] highlights that the Java [Gosling *et al.* 2005] language designers are reluctant to introduce new keywords (as identifiers in legacy programs could then be mistakenly recognised as instances of the new keyword).

Although designers are cautious about changing programming languages, evolution does occur. In this section, two examples of the ways in which programming languages have evolved are discussed. The vocabulary used to describe the scenarios is applicable to evolution of MDE artefacts. Furthermore, MDE sometimes involves the use of general-purpose modelling languages, such as UML [OMG 2007a]. The evolution of general-purpose modelling languages may be similar to that of general-purpose programming languages.

Reduction

Mapping language abstractions to executable concepts can be complicated. Therefore, languages are sometimes evolved to simplify the implementation of translators (compilers, interpreters, etc). It seems that this type of evolution is more likely to occur when language design is a linear process (with a reference implementation occurring after design), and in larger languages.

[Backus 1978] identifies some simplification during FORTRAN's evolution: originally, FORTRAN's `DO` statements were awkward to compile. The semantics of `DO` were simplified such that more efficient object code could be generated from them. Essentially, the simplified `DO` statement allowed linear changes to index statements to be detected (and optimised) by compilers.

The removal of the `RELABEL` construct (which facilitated more straightforward indexing into multi-dimensional arrays) from the FORTRAN language specification [Backus 1978] is a further example of reduction.

Revolution

Developers often form best practices for using languages. Design patterns are one way in which best practices may be communicated with other developers. Incorporating existing design patterns as language constructs is one approach to specifying a new language (e.g. [Bosch 1998]).

Lisp makes idiomatic some of the Fortran List Processing Language (FLPL) design patterns. For example, [McCarthy 1978] describes the awkwardness of using FLPL's `IF` construct, and the way in which experienced developers would often prefer to define a function of the form `XIF(P, T, F)` where `T` was executed iff `P` was true, and `F` was executed otherwise. However, such functions had to be used sparingly, as all three arguments would be evaluated due to the way in which FORTRAN executed function calls. McCarthy [McCarthy 1978] defined a more efficient semantics, wherein `T` (`F`) was only evaluated when `P` was true (false). Because FORTRAN programs could not express these semantics, McCarthy's new construct informed the design of Lisp. Lazy evaluation in functional languages can be seen as a further step on this evolutionary path.

3.2.2 Schema Evolution

This section reviews schema evolution research. Work covering the evolution of XML and database schemata is considered. Both types of schema are used to describe a set of concepts (termed the *universe of discourse* in database literature). Schema designers decide which details of their domain concepts to describe; their schemata provide an abstraction containing only those concepts which are relevant [Elmasri & Navathe 2006, pg. 30]. As such, schemata in these domains may be thought of as analogous to metamodels – they provide a means for describing an abstraction over a phenomenon of interest. Therefore, approaches to identifying, analysing and performing schema evolution are directly relevant to the evolution of metamodels in MDE. However, the patterns of evolution commonly seen in database systems and with XML may be different to those of metamodels because evolution can be:

- **Domain-specific**: Patterns of evolution may be applicable only within a particular domain (e.g. normalisation in a relational database).
- **Language-specific**: The way in which evolution occurs may be influenced by the language (or tool) used to express the change. (For example, some implementations of SQL may not have a `rename relation` command, so alternative means for renaming a relation must be used).

Many of the published works on schema evolution share a similar method, with the aim of defining a taxonomy of evolutionary operators. Schema maintainers are expected to employ these operators to change their schemata. This approach is used heavily in the XML schema evolution community, and was the sole strategy encountered [Guerrini *et al.* 2005, Kramer 2001, Su *et al.* 2001]. Similar taxonomies have been defined for schema evolution in relational database systems (e.g. in [Banerjee *et al.* 1987, Edelweiss & Freitas Moreira 2005]), but other approaches to evolution are also prevalent. One alternative, pro-

posed in [Lerner 2000], is discussed in depth, along with a summary of other work.

XML Schema Evolution

XML provides a specification for defining mark-up languages. XML documents can reference a schema, which provides a description of the ways in which the concepts in the mark-up should relate (i.e. the schema describes the syntax of the XML document). Prior to the definition of the XML Schema specification [W3C 2007a] by the W3C [W3C 2007b], authors of XML documents could use a specific Document Type Definition (DTD) to describe the syntax of their mark-up language. XML Schemata provide a number of advantages over the DTD specification:

- XML Schemata are defined in XML and may, therefore, be validated against another XML Schema. DTDs are specified in another language entirely, which requires a different parser and different validation tools.
- DTDs provide a means for specifying constraints only on the mark-up language, whereas XML Schemata may also specify constraints on the data in an XML document.

Work on the evolution of the structure of XML documents is now discussed. [Guerrini *et al.* 2005] concentrate on changes made to XML Schema, while [Kramer 2001] focuses on DTDs.

[Guerrini *et al.* 2005] propose a set of primitive operators for changing XML schemata. They show this set to be both sound (application of an operator always results in a valid schema) and complete (any valid schema can be produced by composing operators). Their classification also details those operators that are 'validity-preserving' (i.e. application of the operator produces a schema that does not require its instances to be migrated). Guerrini et al. show that the arguments of an operator can influence whether it is validity-preserving. For example, inserting an element is validity-preserving when inclusion of the element is optional for instances of the schema. In addition to soundness and completeness, minimality is another desirable property in a taxonomy of primitive operators for performing schema evolution [Su *et al.* 2001]. To complement a minimal set of primitives, and to improve the conciseness with which schema evolutions can be specified, [Guerrini *et al.* 2005] propose a number of 'high-level' operators, which comprise two or more primitive operators.

[Kramer 2001] provides another taxonomy of primitives for XML schema evolution. To describe her evolution operators, Kramer uses a template, which comprises a name, syntax, semantics, preconditions, resulting DTD changes and resulting data changes section for each operator. This style is similar to a pattern catalogue, but Kramer does not provide a context for her operators

(i.e. there are no examples that describe when the application of an operator may be useful). Kramer utilises her taxonomy in a repository system, Exemplar, for managing the evolution of XML documents and their schemata. The repository provides an environment in which the variation of XML documents can be managed. However, to be of practical use, Exemplar would benefit from integration with a source code management system (to provide features such as branching, and version merging).

As noted in [Pizka & Jürgens 2007], the approaches described in [Kramer 2001, Su *et al.* 2001, Guerrini *et al.* 2005] are complete in the sense that any valid schema can be produced, but do not allow for arbitrary updates of the XML documents in response to schema changes. Hence, none of the approaches discussed in this section ensure that information contained in XML documents is not lost.

Relational Database Schema Evolution

Defining a taxonomy of operators for performing schema updates is also common for supporting relational database schema evolution (e.g. [Edelweiss & Freitas Moreira 2005, Banerjee *et al.* 1987]). However, [Lerner 2000] highlights problems that arise when performing data migration after these taxonomies have been used to specify schema evolution:

> "There are two major issues involved in schema evolution. The first issue is understanding how a schema has changed. The second issue involves deciding when and how to modify the database to address such concerns as efficiency, availability, and impact on existing code. Most research efforts have been aimed at this second issue and assume a small set of schema changes that are easy to support, such as adding and removing record fields, while requiring the maintainer to provide translation routines for more complicated changes. As a result, progress has been made in developing the backend mechanisms to convert, screen, or version the existing data, but little progress has been made on supporting a rich collection of changes" [Lerner 2000, pg. 84].

Fundamentally, [Lerner 2000] believes that any taxonomy of operators for schema evolution is too fine-grained to capture the semantics intended by the schema developer, and therefore cannot be used to provide automated migration: [Lerner 2000] states that existing taxonomies are concerned with the "editing process rather than the editing result". Furthermore, Lerner believes that developing such a taxonomy creates a proliferation of operators, increasing the complexity of specifying migration. To demonstrate, Lerner considers moving a field from one type to another in a schema. This could be expressed using two primitive operators, `delete_field` and `add_field`. However,

the semantics of a `delete_field` command likely dictate that the data associated with the field will be lost, making it unsuitable for use when specifying that a type has been moved. The designer of the taxonomy could introduce a `move_field` command to solve this problem, but now the maintainer of the schema needs to understand the difference between the two ways in which moving a type can be specified, and carefully select the correct one. Lerner provides other examples which elucidate this issue (such as introducing a new type by splitting an existing type). Even though [Lerner 2000] highlights that a fine-grained approach may not be the most suitable for specifying schema evolution, other potential uses for a taxonomy of evolutionary operators (such as being used as a common vocabulary for discussing the restructuring of a schema) are not discussed.

[Lerner 2000] proposes an alternative to operator-based schema evolution in which two versions of a schema are compared to infer the schema changes. Using the inferred changes, migration strategies for the affected data can be proposed. [Lerner 2000] presents algorithms for inferring changes from schemata and performing both automated and guided migration of affected data. By inferring changes, developers maintaining the schema are afforded more flexibility. In particular, they need not use a domain-specific language or editor to change a schema, and can concentrate on the desired result, rather than how best to express the changes to the schema in the small. Furthermore, algorithms for inferring changes have use other than for migration (e.g. for semantically-aware comparison of schemata, similar to that provided by a refactoring-aware *source code management system*, such as [Dig *et al.* 2007]). Comparison of two schema versions might suggest more than one feasible strategy for updating data, and [Lerner 2000] does not propose a mechanism for distinguishing between feasible alternatives.

[Vries & Roddick 2004] propose the introduction of an extra layer to the architecture typical of a relational database management system. They demonstrate the way in which the extra layer can be used to perform migration subsequent to a change of an attribute type. The layer contains (mathematical) relations, termed *mesodata*, that describe the way in which an old value (data prior to migration) maps to one or more new values (data subsequent to migration). These mappings are added to the mesodata by the developer performing schema updates, and are used to semi-automate migration. It is not clear how this approach can be applied when schema evolution is not an attribute type change.

In the O2 database [Ferrandina *et al.* 1995], schema updates are performed using a small domain-specific language. Modification constructs are used to describe the changes to be made to the schema. To perform data migration, O2 provides conversion functions as part of its modification constructs. Conversion functions are either user-defined or default (pre-defined). The pre-defined functions concentrate on providing mappings for attributes whose types are changed (e.g. from a double to an integer; from a set to a list). Additionally,

conversion functions may be executed in conjunction with the schema update, or they may be deferred, and executed only when the data is accessed through the updated schema. Ferrandina et al. observe that deferred updates may prevent unnecessary downtime of the database system. Although the approach outlined in [Ferrandina *et al.* 1995] addresses the concern that "approaches to coping with schema evolution should be concerned with the editing result rather than the editing process" [Lerner 2000], there is no support for some types of evolution such as moving an attribute from one relation to another.

3.2.3 Grammar Evolution

[Klint *et al.* 2003] call for an engineering approach to producing grammarware (grammars and software that depends on grammars, such as parsers and program convertors). The grammarware engineering approach envisaged by Klint et al. is based on best practices and techniques, which they anticipate will be derived from addressing open research challenges. Klint et al. identify seven key questions for grammarware engineering, one of which relates to grammar evolution: "How does one systematically transform grammatical structure when faced with evolution?" [Klint *et al.* 2003, pg. 334].

Between 2001 and 2005, Ralf Lämmel (an author of [Klint *et al.* 2003]) and his colleagues at Vrije Universiteit published several important papers on grammar evolution. [Lämmel 2001] proposes a taxonomy of operators for semi-automatic grammar refactoring and demonstrates their usefulness in recovering the formal specifications of undocumented grammars (such as VS COBOL II in [Lämmel & Verhoef 2001]) and in specifying generic refactorings [Lämmel 2002].

The work of Lämmel et al. focuses on grammar evolution for refactoring or for *grammar recovery* (corrective evolution in which a deviation from a language reference is removed), but does not address the impact of grammar evolution on corresponding programs or grammarware. For instance, when a grammar changes, updates are potentially required both to programs written in that grammar and to tools that parse, generate or otherwise manipulate programs written in that grammar.

[Pizka & Jürgens 2007] recognise and seek to address the challenge of grammar-program co-evolution. Pizka and Juergens believe that most grammars evolve over time and that, without tool support, co-evolution is a complex, time-consuming and error prone task. To this end, [Pizka & Jürgens 2007] proposes Lever, a language evolution tool, which defines and uses operators for changing grammars (and programs) in an approach that is inspired by [Lämmel 2001].

Compared to the taxonomy in [Lämmel 2001], Lever can be used to manage the evolution of grammars, programs and the co-evolution of grammars and programs, and the taxonomy defined by Lämmel et al. can be used only to manage grammar evolution. However, as a consequence, Lever sacrifices the formal preservation properties of the taxonomy defined by Lämmel et al.

3.2.4 Evolution of MDE Artefacts

As discussed in Chapter 1, the evolution of development artefacts during MDE inhibits the productivity and maintainability of model-driven approaches for constructing software systems. Mitigating the effects of evolution on MDE is an open research topic, to which this thesis contributes.

This section discusses literature that explores the evolution of development artefacts used when performing MDE. [Deursen *et al.* 2007] highlight that evolution in MDE is complicated, because it spans multiple dimensions. In particular, there are three types of development artefact specific to MDE: models, metamodels, and specifications of model management tasks[2]. A change to one type of artefact can affect other artefacts (possibly of a different type).

[Sprinkle & Karsai 2004] highlights that the evolution of an artefact can appear to be either *syntactic* or *semantic*. In the former, no information is known about the intention of the the evolutionary change. In the latter, a lack of detailed information about the semantics of evolution can reduce the extent to which change propagation can be automated. For example, consider the case where a class is deleted from a metamodel. The following questions typically need to be answered to facilitate evolution:

- Should subtypes of the deleted class also be removed? If not, should their inheritance hierarchy be changed? What is the correct type for references that used to have the type of the deleted class?

- Suppose that the evolving metamodel was the target of a previous model-to-model transformation. Should the data that was previously transformed to instances of the deleted class now be transformed to instances of another metamodel class?

- What should happen to instances of the deleted metamodel class? Perhaps they should be removed too, or perhaps their data should be migrated to new instances of another class.

Tools that recognise only syntactic evolution tend to lack the information required for full automation of evolution activities. Furthermore, tools that focus only upon syntax cannot be applied in the face of additive changes [Gruschko *et al.* 2007]. There are complexities involved in recording the semantics of software evolution. For example, the semantics of an impacted artefact need not always be preserved: this is often the case in corrective evolution.

Notwithstanding the challenges described above, MDE has great potential for managing software evolution and automating software evolution activities, particularly because of model transformations (Section 2.1.4). Approaches for

[2]Some examples of model management tasks include model-to-model transformation, model-to-text transformation, model validation, model merging and model comparison.

managing evolution in other fields, described above, must consider the way in which artefacts are updated when changes are propagated from one artefact to another. Model transformation languages already fulfil this role in MDE. In addition, model transformations provide a (limited) form of traceability between MDE artefacts, which can be used in impact analysis.

This section focuses on the three types of evolution most commonly discussed in model-driven engineering literature. *Model refactoring* is used to improve the quality of a model without changing its functional behaviour. *Model synchronisation* involves updating a model in response to a change made in another model, usually by executing a model-to-model transformation. *Model-metamodel co-evolution* involves updating a model in response to a change made to a metamodel. This section concludes by reviewing existing techniques for visualising model-to-model transformation and assessing their usefulness for understanding evolution in the context of MDE.

Model Refactoring

Refactoring (Section 3.1.2) is a perfective software evolution activity in which the structure and representation of a system is improved without changing its functional behaviour. Refactoring has been studied in the context of MDE because refactoring can be domain-specific (e.g. normalisation in relational databases). Model refactoring languages allow metamodel developers to capture commonly occurring refactoring patterns and provide their users with model editors that support automatic refactoring.

In model transformation terminology (discussed in Section 2.1.4), a refactoring is an *endogenous*, *in-place* transformation. Refactorings are applied to an artefact (e.g. model, code) producing a semantically equivalent artefact, and hence an artefact that conforms to the same rules and structures as the original. Because refactorings are used to improve the structure of an existing artefact, the refactored artefact typically replaces the original. Endogenous, in-place transformation languages, suitable for refactoring, are described in [Biermann *et al.* 2006, Porres 2003] (which propose declarative approaches based on graph theory) and in [Kolovos *et al.* 2007a] (which proposes mixing declarative and imperative constructs).

There are similarities between the structures defined in the MOF metamodelling language and in object-oriented programming languages. For the latter, refactoring pattern catalogues exist (such as [Fowler 1999]), which might usefully be applied to modelling languages. [Moha *et al.* 2009] provides a notation for specifying refactorings for MOF and UML models and Java programs in a generic (metamodel-independent) manner. Because MOF, UML and the Java language share some concepts with the same semantics (such as classes and attributes), [Moha *et al.* 2009] show that refactorings can be shared among them, but only consider 3 of the object-oriented refactorings identified in [Fowler 1999]. To more thoroughly understand metamodel-

independent refactoring, a larger number of refactorings and languages should be explored.

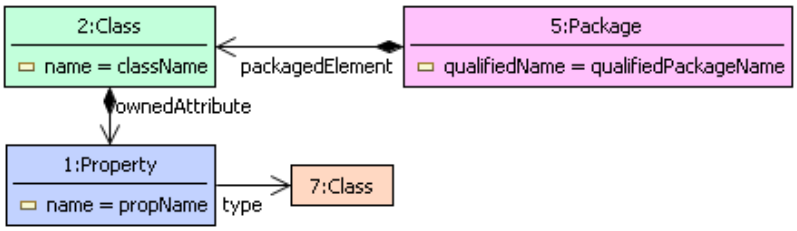
Abstraction is a fundamental benefit of MDE (Section 2.5.1). Defining a domain-specific language is one way in which abstraction can be realised for MDE (Section 2.4.1). In addition to tools for defining modelling languages, generating model editors and performing model transformation, model-driven development environments might benefit from mechanisms for defining domain-specific refactorings. In particular, metamodel developers may wish to document common patterns of evolution, perhaps in an executable format.

Eclipse, an extensible development environment, provides a library for building development tools for textual languages, LTK (language toolkit) [Frenzel 2006]. LTK allows developers to specify – in Java – refactorings for their language, which can be invoked via the language editor. LTK makes no assumptions on the way in which languages will be structured, and as such refactoring code that operates on models must interact with the modelling framework directly.

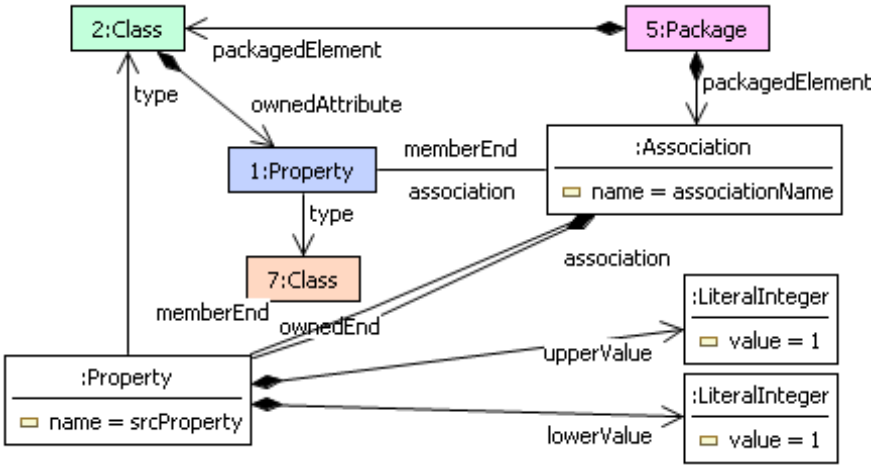
The Epsilon Wizard Language (EWL) [Kolovos *et al.* 2007a] is a model transformation language tailored for the specification of model refactorings. EWL is built atop Epsilon and its object language (EOL), which can query, update and navigate models represented in a diverse range of modelling technologies (Section 2.3.2). Consequently, EWL, unlike LTK, abstracts over modelling frameworks.

[Arendt *et al.* 2009] present EMF Refactor, comparing it with EWL and the LTK by specifying a refactoring on a UML model. EMF Refactor, like EWL, contributes a model transformation language tailored for refactoring. In contrast to EWL, EMF Refactor has a visual (rather than textual) syntax, and is based on graph transformation concepts. Figure 3.2 shows the "Change attribute to association end" refactoring for the UML metamodel in EMF Refactor. The left-hand side of the refactoring rule (Figure 3.2(a)) matches a Class whose owned attributes contains a Property whose type has the same name as a Class. The right-hand side of the rule (Figure 3.2(b)) introduces a new Association, whose member end is the Property matched in the left-hand side of the rule. Due to the visual syntax, EMF Refactor might be usable only with modelling technologies based on MOF (which has a graphical concrete-syntax based on UML class diagrams). From [Arendt *et al.* 2009], it is not clear to what extent EMF Refactor can be used with modelling technologies other than EMF.

[Kolovos *et al.* 2007a] and [Arendt *et al.* 2009] focus on refactoring a model in isolation. Neither approach can be used to specify *inter-model refactorings*, which impact more than one model at once. The Eclipse Java Development Tools support refactorings of Java code that update many source-code artefacts at once: for example, renaming a class in one source file updates references to that class in other source files. In the context of MDE, support for



(a) Left-hand side matching rule.



(b) Right-hand side production rule.

Figure 3.2: Attribute to association end refactoring in EMF Refactor. Taken from [Arendt *et al.* 2009].

inter-model refactoring would facilitate a greater degree of model modularisation, regarded by [Kolovos *et al.* 2008c] as a solution to scalability, one of the challenges faced by MDE.

According to [Mens *et al.* 2007], "research in model refactoring is still in its infancy." Mens et al. identify formalisms for investigating the feasibility and scalability of model refactoring. In particular, Mens et al. suggest that meaning-preservation (an objective of refactoring, as discussed in Section 3.1.2) can be checked by evaluating OCL constraints, behavioural models or downstream program code.

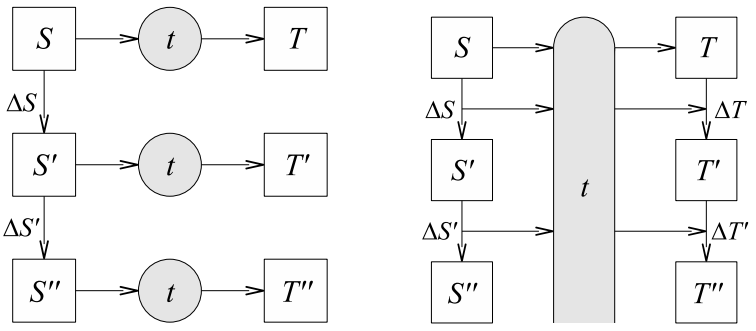
Model Synchronisation

Changes made to development artefacts may require the *synchronisation* of related artefacts (models, code, documentation). Traceability links (which capture the relationships of software artefacts) facilitate synchronisation. This

section discusses the way in which change propagation is approached in the literature, which typically involves using an incremental style of transformation. Work that addresses more fundamental aspects of model synchronisation, such as capturing trace links and performing impact analysis are also discussed. Finally, synchronisation between models and text and between models and trace links is also considered.

Incremental Transformation Many model synchronisation approaches extend or instrument existing model-to-model transformation languages. Declarative transformation languages lend themselves to the specification of bidirectional transformations (which [Fritzsche *et al.* 2008] describe as traceability-by-design) and *incremental transformations*, a style of model transformation that facilitates incremental updates of the target model. In fact, most model synchronisation literature focuses on incremental transformation.

Incremental transformation is most often achieved in one of two ways. Because model-to-model transformation is used to generate one or more target models from one or more source models, when a source model changes, the model-to-model transformation can be invoked to completely re-generate the target models. [Hearnden *et al.* 2006] call this activity *re-transformation*, and propose an alternative approach, *live transformation*, in which the transformation context is persistent. Figure 3.3 illustrates the differences between re transformation and live transformation, showing the evolution of source and target models on the left-hand and right-hand sides, respectively, and the transformation context in the middle. Live transformation facilitates change propagation from the source to the target models without completely re-generating the target models and is therefore a more efficient approach. As well as in [Hearnden *et al.* 2006], live transformation is used to achieve incremental transformation in [Ráth *et al.* 2008] and [Tratt 2008].



(a) Re-transformation. (b) Live transformation.

Figure 3.3: Approaches to incremental transformation. Taken from [Hearnden *et al.* 2006].

Primarily, incremental transformation has been used to address the scalability of model transformations. For large models, transformation execution time has been shown to be significantly reduced by using incremental transformation [Hearnden *et al.* 2006]. However, [Kolovos *et al.* 2008c] suggests that scalability should be addressed not only by attempting to develop techniques for increasing the speed of model transformation, but also by providing principles, practices and tools for building models that are less monolithic and more modular. For this end, model synchronisation research that focuses solely on increasing scalability is unhelpful and should instead focus on improving maintainability in conjunction with – or rather than – scalability.

Model synchronisation and incremental transformation can be applied to decouple models and facilitate greater modularisation, although this is not commonly discussed in the literature. [Fritzsche *et al.* 2008] describe an automated, model-driven approach to performance engineering. Fritzche et al. contribute a transformation that produces, from any UML model, a model for which performance characteristics can be readily analysed. The relationships between UML and performance model artefacts are recorded using traceability links. The results of the performance analysis are later fed back to the UML model using an incremental transformation made possible by the traceability links. Using this approach, performance engineers can focus primarily on the performance models, while other engineers are shielded from the low-level detail of the performance analysis. As such, Fritzsche et al. show that two different modelling concerns can be separated and decoupled, yet remain cohesive via the application of model synchronisation.

Towards automated model synchronisation Some existing work provides a foundation for automating model synchronisation activities. Theoretical aspects of the traceability literature were reviewed in Section 3.1.4, and explored the automated activities that traceability facilitates, such as impact analysis and change propagation. This section now analyses the traceability research in the context of model-driven engineering and focuses on the way in which traceability facilitates the automation of model synchronisation activities.

Aside from live transformation, other techniques for capturing trace links between models have been reported. Enriching a model-to-model transformation with traceability information is discussed in [Jouault 2005], which contributes a generic higher-order transformation for this purpose. Given a transformation, the generic higher-order transformation adds transformation rules that produce a traceability model. In contrast to the genericity of the approach described in [Jouault 2005], [Drivalos *et al.* 2008] propose domain-specific traceability metamodels for richer traceability link semantics. Further research is required to assess the requirements of automated model synchronisation tools and to select appropriate traceability approaches for their im-

plementation.

Impact analysis is used to reason about the effects of a change to a development artefact. As well as facilitating change propagation, impact analysis can help to predict the cost and complexity of changes [Bohner 2002]. Impact analysis requires solutions to several sub-problems, which include change detection, analysis of the effects of a change, and effective display of the analysis.

[Briand *et al.* 2003] contributes an impact analysis tool for UML models that compares original and evolved versions of the same model, producing a report of evolved model elements that have been impacted by the changes to the original model elements. To facilitate the impact analysis, [Briand *et al.* 2003] identifies change patterns that comprise, among other properties, a trigger (for change detection) and an impact rule (for marking model elements affected by this change). Figure 3.4 shows a sample impact analysis pattern for UML sequence diagrams, which is triggered when a message is added, and marks the sending class, the sending operation and the postcondition of the sending operation as impacted.

[Winkler & Pilgrim 2009] note that only event-based approaches, such as the one described in [Briand *et al.* 2003], have been proposed for automating impact analysis. Because of the use of patterns for detecting changes and determining reactions, event-based impact analysis is similar to differencing approaches for schema evolution (for example, [Lerner 2000], which was discussed in Section 3.2.2). When more than one trigger might apply, event-based impact analysis approaches must provide mechanisms for selecting between applicable patterns. In [Briand *et al.* 2003], the selection policy is implicit (cannot be changed by the user) and further analysis is needed to assess its limitations.

Finally, model synchronisation tools might apply techniques used in automated synchronisation tools for traditional development environments, such as the refactoring functionality of the Eclipse Java Development Tools [Fuhrer *et al.* 2007].

Synchronisation of models with text and trace links So far, this section has concentrated on model-to-model synchronisation, which is facilitated by traceability. Traceability is important for other software evolution activities in a model-driven development environment – such as synchronisation between models and text and between models and trace links – and these activities are now discussed.

While most of the model synchronisation literature focuses on synchronising models with other models, some papers consider synchronisation between models and other types of artefact. For synchronising changes in requirements documents with models, there is abundance of work in the field of requirements engineering, where the need for traceability was first reported. For synchronising models with generated text (during code generation, for example), the model-to-text language, Epsilon Generation Language (EGL)

Change Title: Changed Sequence Diagram – Added Message

Change Code: CSDVAM

Changed Element: `model::behaviouralElements::collaborations::SequenceDiagramView`

Added Property: `model::behaviouralElements::collaborations::Message`

Impacted Elements:
`model::foundation::core::ClassClassifier`
`model::foundation::core::Operation`
`model::foundation::core::Postcondition`

Description: The base class of the classifier role that sends the added message is impacted. The operation that sends the added message is impacted and its postcondition is also impacted.

Rationale: The sending/source class now sends a new message and one of its operations, actually sending the added message, is impacted. This operation is known or not, depending on whether the message triggering the added message corresponds to an invoked operation. If, for example, it is a signal then we may not know the operation, just by looking at the sequence diagram. The impacted postcondition may now not represent the effect (what is true on completion) of its operation.

Resulting Changes: The implementation of the base class may have to be modified. The method of the impacted operation may have to be modified. The impacted postcondition should be checked to ensure that it is still valid.

Invoked Rule: Changed Class Operation – Changed Postcondition (CCOCPst)

OCL Expressions:

```
context modelChanges::Change def:
    let addedMessage:Message = self.changedElement.oclAsType(SequenceDiagramView).
        Message->select(m:Message | m.getIDStr()=self.propertyID)
    let sendingOperation:Operation = (
        if addedMessage.activator.action.oclIsTypeOf(CallAction) then
            addedMessage.sender.base.operation->select(o:Operation |
            o.equals(addedMessage.activator.callAction.operation))
        else
            null
        endif)
context modelChanges::Change – class
    addedMessage.sender.base
context modelChanges::Change – operation
    sendingOperation
context modelChanges::Change – postcondition
    sendingOperation.postcondition
```

Figure 3.4: Exemplar impact analysis pattern, taken from [Briand *et al.* 2003].

[Rose *et al.* 2008], produces traceability links between code generation templates and generated files. Sections of code can be marked protected, and are not overwritten by subsequent invocations of the code generation template. As described in [Olsen & Oldevik 2007], the MOFScript model-to-text language, like EGL, provides protected sections and, unlike EGL, also stores traceability links in a structured manner. The traceability links described in [Olsen & Oldevik 2007] can be used for impact analysis, model coverage (for highlighting which areas of the model contribute to the generated code) and orphan analysis (for detecting invalid traceability links).

Trace links can be affected when development artefacts change. Synchronisation tools rely on accurate trace links and hence the maintenance of trace links is important. [Winkler & Pilgrim 2009] suggests that trace versioning should be used to address the challenges of trace link maintenance, which

include the accidental inclusion of unintended dependencies as well as the exclusion of necessary dependencies. Furthermore, [Winkler & Pilgrim 2009] notes that, although versioning traces has been explored in specialised areas (such as hypermedia [Nguyen *et al.* 2005]), there is no holistic approach for versioning traces.

Model-metamodel Co-Evolution

A metamodel describes the structures and rules for a family of models. When a model uses the structures and adheres to the rules defined by a metamodel, the model is said to *conform* to the metamodel [Bézivin 2005]. A change to a metamodel might require changes to models to ensure the preservation of conformance. The process of evolving a metamodel and its models together to preserve conformance is termed *model-metamodel co-evolution* and is subsequently referred to as *co-evolution*. This section explores existing approaches to co-evolution, comparing them with work from the closely related areas of schema and grammar evolution approaches (Sections 3.2.2 and 3.2.3). A more thorough analysis of co-evolution approaches is conducted in Chapter 4.

Co-evolution theory A co-evolution process involves changing a metamodel and updating instance models to preserve conformance. Often, the two activities are considered separately, and the latter is termed *migration*. In this thesis, the term *migration strategy* is used to mean an algorithm that specifies migration. [Sprinkle & Karsai 2004] were the first to identify the need for approaches that consider the specific requirements of co-evolution, treating it separately from other development artefacts. In particular, Sprinkle and Karsai describe migration as distinct from – and as having unique challenges compared to – the more general activity of model-to-model transformation. [Sprinkle 2003] uses the phrase "evolution, not revolution" to highlight and emphasise that, during co-evolution, the difference between source and target metamodels is often small.

Understanding the situations in which co-evolution must be managed is important for formulating the requirements for co-evolution tools. However, co-evolution literature rarely reports on the ways in which co-evolution is managed in practice. [Herrmannsdoerfer *et al.* 2009b] reports that migration is sometimes made unnecessary by evolving a metamodel such that the conformance of models is not affected (for example, making only additive changes). [Cicchetti *et al.* 2008] suggests that co-evolution can be carried out by more than one person, and that metamodel developers and model users might not know one another.

Co-evolution patterns Much of the co-evolution literature suggests that the way in which migration is performed should vary depending on the type of metamodel changes made [Gruschko *et al.* 2007, Herrmannsdoerfer *et al.* 2009b,

Cicchetti *et al.* 2008, Garcés *et al.* 2009]. In particular, the co-evolution literature identifies two important classifications of metamodel changes that affect the way in which migration is performed. [Gruschko *et al.* 2007] classify metamodel changes, recognising that, depending on the type of metamodel change, migration might be unnecessary (*non-breaking* change), can be automated (*breaking and resolvable* change) and can be automated only when guided by a developer (*breaking and non-resolvable* change). [Herrmannsdoerfer *et al.* 2008] classify metamodel changes into *metamodel-independent* (observed in the evolution of more than one metamodel) and *metamodel-specific* (observed in the evolution of only one metamodel).

Further research is needed to identify categories of metamodel changes because automated co-evolution approaches are built atop them. [Herrmannsdoerfer *et al.* 2008] suggests that a large fraction of metamodel changes re-occur, but the study considers only two metamodels, both taken from the same organisation. Assessing the extent to which changes re-occur across a larger and broader range of metamodels is an open research challenge to which this thesis contributes, particularly in Chapter 4.

Co-evolution approaches Several approaches for managing co-evolution have been proposed, most of which are based on one of the two classifications of metamodel changes described above.

Re-use of migration knowledge is a primary concern in the work of Herrmannsdöerfer. [Herrmannsdoerfer *et al.* 2008] describes an empirical study of the history of two metamodels from the automobile industry, observing that a large number of metamodel changes re-occur. [Herrmannsdoerfer *et al.* 2009b] proposes a co-evolution tool, COPE, that provides a library of co-evolutionary operators. Operators are applied to evolve a metamodel and have pre-defined migration semantics. The application of each operator is recorded, and used to generate an executable migration strategy. Due to its use of re-usable operators, COPE shares characteristics with operator-based approaches for schema and grammar evolution (Sections 3.2.2 and 3.2.3). Consequently, the limitations for operator-based schema evolution approaches identified in [Lerner 2000] apply to COPE. Balancing expressiveness and understandability is a key challenge for operator-based approaches because the former implies a large number of operators while the latter a small number of operators.

[Gruschko *et al.* 2007] suggest inferring co-evolution strategies, based on either a difference model of two versions of the evolving metamodel (direct comparison) or on a list of changes recorded during the evolution of a metamodel (indirect comparison). To this end, [Gruschko *et al.* 2007] contributes a *metamodel matching* co-evolution process, shown in Figure 3.5.

Both [Cicchetti *et al.* 2008] and [Garcés *et al.* 2009] extend the work of [Gruschko *et al.* 2007], using the metamodel matching co-evolution process.

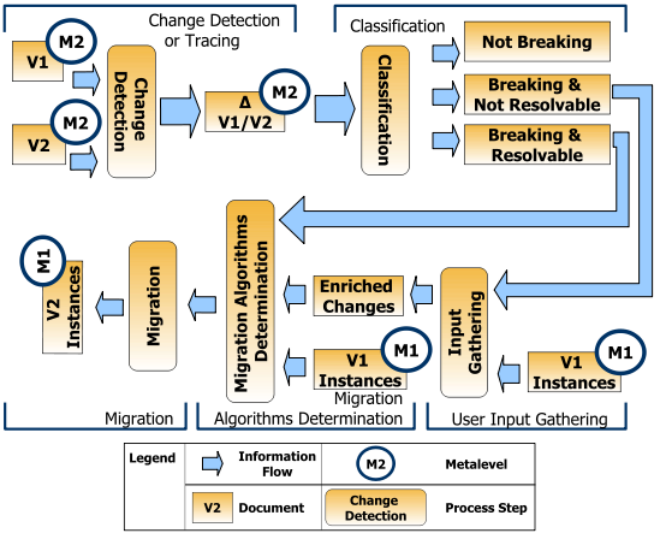


Figure 3.5: The co-evolution process described in [Gruschko *et al.* 2007].

Both also use higher-order model transformation[3] for determining the migration strategy (the penultimate phase in Figure 3.5). [Cicchetti *et al.* 2008] contributes a metamodel for describing the similarities and differences between two versions of a metamodel, enabling a model-driven approach to generating model migration strategies. [Garcés *et al.* 2009] provides a similar metamodel, but uses a metamodel matching process that can be customised by the user, who specifies matching heuristics to form a matching strategy. Otherwise, the co-evolution approaches described in [Cicchetti *et al.* 2008] and [Garcés *et al.* 2009] are fully automatic and cannot be guided by the user. Clearly then, accuracy is important for approaches that compare two metamodel versions, but the co-evolution literature does not assess the extent to which approaches like [Cicchetti *et al.* 2008] and [Garcés *et al.* 2009] can be applied.

Some co-evolution approaches predate the classifications of metamodel changes described above. For instance, [Wachsmuth 2007] proposes a preliminary catalogue of metamodel changes and was the first to employ higher-order transformation for specifying model migration. However, [Wachsmuth 2007] considers a small number of metamodel changes occurring in isolation and, as such, it is not clear whether the approach can be used in general. [Sprinkle 2003] proposes a visual transformation language for specifying model migration, based on graph transformation theory. As such, the migration language proposed by Sprinkle is less expressive than imperative or hybrid transformation

[3] A model-to-model transformation that consumes or produces a model-to-model transformation is *higher-order*.

languages (as discussed in Section 2.1.4).

Summary Automated migration is still an open research challenge. Co-evolution approaches are in their infancy, and key problems need to be addressed. For example, [Lerner 2000] notes that matching schemas (metamodels) can yield more than one feasible set of migration strategies. [Cicchetti *et al.* 2008] does not acknowledge this challenge. [Garcés *et al.* 2009] offers heuristics for controlling metamodel matching, which might affect the predictability of the co-evolution process.

Another open research challenge is in identifying an appropriate notation for describing migration. [Wachsmuth 2007, Cicchetti *et al.* 2008] use higher-order transformations, while [Herrmannsdoerfer *et al.* 2009b] uses a general-purpose programming language. Because migration is a specialisation of model-to-model transformation [Sprinkle & Karsai 2004], languages other than model-to-model transformation languages might be more suitable for describing migration.

Until co-evolution tools reach maturity, improving MDE modelling frameworks to better support co-evolution is necessary. For example, the Eclipse Modelling Framework [Steinberg *et al.* 2008] cannot load models that no longer conform to their metamodel and, hence non-conformant models cannot be used for model-driven development with EMF.

Visualisation

To better understand the effects of evolution on development artefacts, visualising different versions of each artefact may be beneficial. Existing research for comparing text can be enhanced to perform semantic-differencing of models with a textual concrete syntax. For models with a visual concrete syntax, another approach is required.

[Pilgrim *et al.* 2008] have implemented a three-dimensional editor for exploring transformation chains (the sequential composition of model-to-model transformations). Their tool enables developers to visualise the way in which model elements are transformed throughout the chain. Figure 3.6 depicts a sample transformation chain visualisation. Each plane represents a model. The links between each plane illustrates the effects of a model-to-model transformation.

The visualisation technology described in [Pilgrim *et al.* 2008] could be used to facilitate exploration of artefact evolution.

3.3 Summary

This chapter reviews and analyses software evolution literature, introducing terminology and describing *impact analysis* and *change propagation*, two evolution activities explored in the remainder of this thesis. Principles and prac-

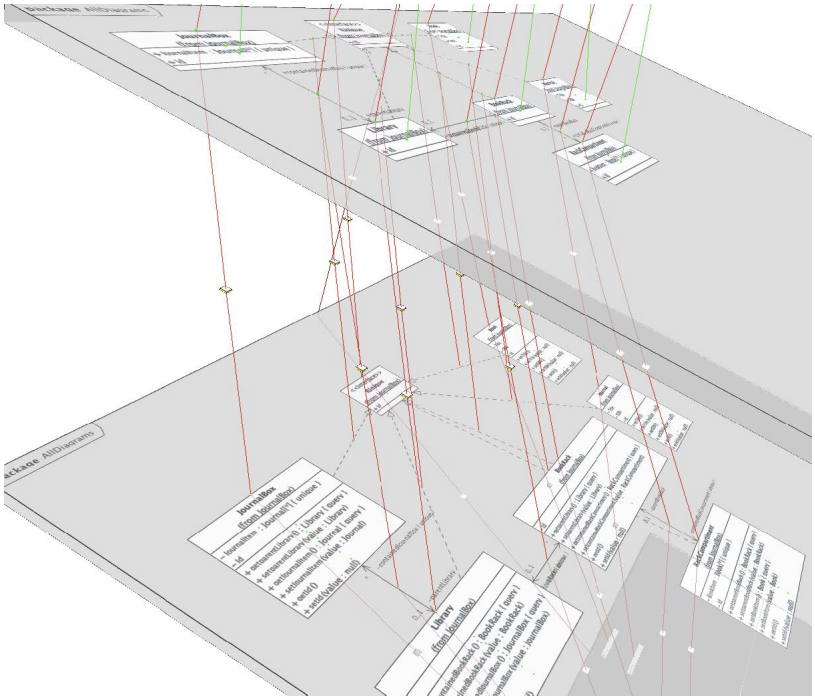


Figure 3.6: Visualising a transformation chain. Taken from [Pilgrim *et al.* 2008].

tices of software evolution (from the fields of programming languages, relational database systems and grammarware) were compared, contrasted and analysed. In particular, software evolution literature from the MDE community was reviewed and analysed to allow the formulation of potential research directions for this thesis. The chapter now concludes by synthesising, from the reviewed literature, research challenges for managing software evolution in the context of MDE.

Model Refactoring Challenges The model refactoring literature propose tools and techniques for improving the quality of existing models without affecting their functional behaviour. In traditional development environments, inter-artefact refactoring (in which changes span more than one development artefact) is often automated, but none of the model refactoring papers discussed in this chapter consider inter-model refactoring. In general, the refactoring literature covers several concerns, such as identification, validation and assessment (Section 3.1.2), but the model refactoring literature considers only the specification and application of refactoring. To better understand the costs and benefits of model refactoring, further model refactoring research must consider all of the concerns considered in the refactoring literature in

general.

Model Synchronisation Challenges Improved scalability is the primary motivation of most model synchronisation research. However, [Fritzsche *et al.* 2008] suggest that model synchronisation can be used to improve the maintainability of a system via modularisation. Building on the work by [Fritzsche *et al.* 2008], further research should explore the extent to which model synchronisation can be used to manage evolution. [Winkler & Pilgrim 2009] observe that, for impact analysis between models, only event-based approaches have been reported; other approaches – used successfully to manage evolution in other fields (such as relational databases and grammarware) – have not been applied. Few papers consider synchronisation with other artefacts and maintaining trace links and there is potential for further research in these areas.

Model-Metamodel Co-evolution Challenges To better understand model-metamodel co-evolution, further studies of the ways in which metamodels change are required. [Herrmannsdoerfer *et al.* 2008] report an empirical study of industrial metamodels, but focus only on two metamodels produced in the same organisation. Challenges for co-evolution reported in other fields have not been addressed by the model-metamodel co-evolution literature. For example, [Lerner 2000] notes that comparing two versions of a changed artefact (such as metamodel) can suggest more than one feasible migration strategy. Approaches to co-evolution that do not consider the way in which a metamodel has changed, such as [Cicchetti *et al.* 2008, Garcés *et al.* 2009] must address this challenge. A range of notations are used for model migration, including model-to-model transformation languages and general-purpose programming languages, which is a challenge for the comparison of co-evolution tools. Finally, contemporary MDE modelling frameworks do not facilitate MDE for non-conformant models, which is problematic at least until co-evolution tools reach maturity.

General Challenges for Evolution in MDE From the analysis in this chapter, several research challenges for software evolution in the context of MDE are apparent. Greater understanding of the situations in which evolution occurs informs the identification and management of evolution, yet few papers study evolution in real-world MDE projects. Analysis of existing projects can yield patterns of evolution, providing a common vocabulary for thinking and communicating about evolution. Evolution notations and tools are built atop these patterns to automate some evolution activities. Few papers that consider evolution focus less on evolution patterns than on evolution tools and their implementation. Finally, recording, analysing and visualising changes made over the long term to MDE development artefacts and to MDE projects is an area that is not considered in the literature.

As well as directing the thesis research, the above challenges influenced the choice of research method. Most of the software evolution research discussed in this chapter uses a similar method: first, identify and categorise evolutionary changes by considering all of the ways in which artefacts can change. Next, design a taxonomy of operators that capture these changes or a matching algorithm that detects the application of the changes. Then, implement a tool for applying operators, invoking a matching algorithm, or trigger change events. Finally, evaluate the tool on existing projects containing examples of evolution.

The research in this thesis follows a different method, based on the method used by Digg in his work on program refactoring ([Dig & Johnson 2006b, Dig & Johnson 2006a, Dig *et al.* 2006, Dig *et al.* 2007]). First, existing projects are analysed to better understand the situations in which evolution occurs. From this analysis, research requirements are derived, and structures and process for managing evolution are implemented. The structures and process are evaluated by comparison with related work and by application on an existing project in which there is a need to manage evolution.

Using the literature reviewed and the research challenges identified in this chapter, Chapter 4 analyses examples of evolution from existing MDE projects and derives requirements for structures and processes for managing evolution in the context of MDE.

Chapter 4

Analysis

The literature review presented in Chapter 3 motivated a deeper analysis of existing techniques for managing and identifying the effects of evolution in the context of MDE. Figure 4.1 summarises the objectives of this chapter. On the left are the artefacts used to produce those on the right. As shown in Figure 4.1, examples of evolution in MDE projects were located (Section 4.1) and used to analyse existing co-evolution techniques. Analysis led to a categorisation and comparison of existing co-evolution approaches (Section 4.2) and to the identification of modelling framework characteristics that restrict the way in which co-evolution can be managed (Section 4.2.1). Research requirements for this thesis were identified from the analysis presented in this chapter (Section 4.3).

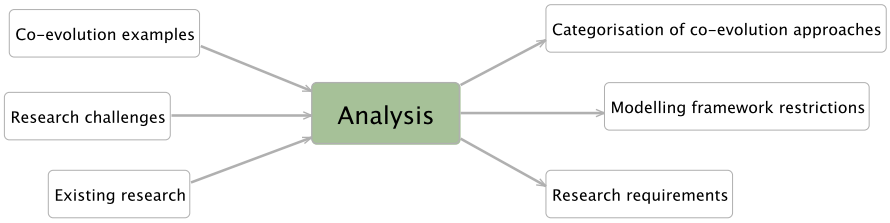


Figure 4.1: Analysis chapter overview.

Earlier in this thesis, the term *modelling framework* has meant an implementation of a set of abstractions for defining, checking and otherwise managing models. The remainder of this thesis focuses on modelling frameworks used for MDE, and, more specifically, modern MDE modelling frameworks such as the Eclipse Modeling Framework [Steinberg *et al.* 2008]. Therefore, the term *modelling framework* is used to mean modern MDE modelling frameworks, unless otherwise stated.

4.1 Locating Data

In Chapter 3, three categories of evolutionary change were identified: model refactoring, synchronisation and co-evolution. Existing MDE projects were examined for examples of synchronisation and co-evolution and, due to time constraints, examples of model refactoring were not considered. The examples were used to provide requirements for developing structures and processes for evolutionary changes in the context of MDE. In this section, the requirements used to select example data are described, along with candidate and selected MDE projects. The section concludes with a discussion of further examples, which were obtained from joint research – with colleagues in this department and at the University of Kent – and from related work on the evolution of object-oriented programs.

4.1.1 Requirements

The requirements used to select example data are now discussed. Requirements were partitioned into: those necessary for studying each of the two categories of evolutionary change, and common requirements (applicable to both categories of evolutionary change). MDE projects were evaluated against these requirements, and several were selected for further analysis.

Common requirements

Every candidate project needs to use MDE. Specifically, both metamodelling and model transformation must be used (requirement R1). In addition, each candidate project needs to provide historical information to trace the evolution of development artefacts (R2). For example, several versions of the project are needed perhaps in a source code management system. Finally, a candidate project needs to have undergone a number of significant changes[1] (R3).

Co-evolution requirements

A candidate project for the study of co-evolution needs to define a metamodel and some changes to that metamodel (R4). In the projects considered, the metamodel changes took the form of either another version of the metamodel, or a history (which recorded each of the steps used to produce the adapted metamodel). A candidate project also needs to provide example instances of models before and after each migration activity (R5).

Ideally, a candidate project should include more than one metamodel adaptation in sequence, so as to represent the way in which the same development artefacts continue to evolve over time (optional requirement O1).

[1] This is deliberately vague. Further details are given in Section 4.1.2.

Table 4.1: Candidates for study of evolution in existing MDE projects

Name	Requirements									
	Common			Co-evolution			Synchronisation			
	R1	R2	R3	R4	R5	O1	R6	R7	R8	O2
GSN	x			x						
OMG	x			x			x			
Zoos	x	x		x						
MDT	x	x		x		x				
MODELPLEX	x	x	x	x		x	x	x		
FPTC	x	x	x	x	x					
xText	x	x	x	x	x	x	x	x		x
GMF	x	x	x	x	x	x	x	x		x

Synchronisation requirements

A candidate project for the study of synchronisation needs to define a model-to-model transformation (R6). Furthermore, a candidate project has to include many examples of source and target models for that transformation (R7). A candidate project needs to provide many examples of the kinds of change (to either source or target model) that cause inconsistency between the models (R8).

Ideally, a candidate project should also include transformation chains (more than one model-to-model transformation, executed sequentially) (O2). Chains of transformations are prescribed by the MDA guidelines [Kleppe *et al.* 2003].

4.1.2 Project Selection

Eight candidate projects were considered for the study. Table 4.1 shows which of the requirements are fulfilled by each of the candidates. Each candidate is now discussed in turn.

GSN

Georgios Despotou and Tim Kelly, members of this department's High Integrity Systems Engineering group, are constructing a metamodel for Goal Structuring Notation (GSN). The metamodel has been developed incrementally. There is no accurate and detailed version history for the GSN metamodel (requirement R2). **Suitability for study:** Unsuitable.

OMG

The Object Management Group (OMG) [OMG 2008b] oversees the development of model-driven technologies. The Vice President and Technical Director of OMG, Andrew Watson, references the development of two MDE projects in

[Watson 2008]. Personal correspondence with Watson ascertained that source code is available for one of the projects, but there is no version history. **Suitability for study:** Unsuitable.

Zoos

A zoo is a collection of metamodels, authored in a common metamodelling language. I considered two zoos (the Atlantic Zoo and the AtlantEcore Zoo[2]), but neither contained any significant external metamodel changes. Those changes that were made involved only renaming of meta-classes (trivial to migrate) or additive changes (which do not affect consistency, and therefore require no migration). **Suitability for study:** Unsuitable.

MDT

The Eclipse Model Development Tools (MDT) [Eclipse 2009a] provides implementations of industry-standard metamodels, such as UML2 [OMG 2007a] and OCL [OMG 2006]. Like the metamodel zoos, the version history for the MDT metamodels contained no significant changes. **Suitability for study:** Unsuitable.

MODELPLEX

Jendrik Johannes, a research assistant at TU Dresden, has made available work from the European project, MODELPLEX[3]. Johannes's work involves transforming UML models to Tool Independent Performance Models (TIPM) for simulation. Although the TIPM metamodel and the UML-to-TIPM transformation have been changed significantly, no significant changes have been made to the models. The TIPM metamodel was changed such that conformance was not affected. **Suitability for study:** Unsuitable.

FPTC

Failure Propagation and Transformation Calculus (FPTC), developed by Malcolm Wallace in this department, provides a means for reasoning about the failure behaviour of complex systems. In an earlier project, Richard Paige and I developed an implementation of FPTC in Eclipse. The implementation includes an FPTC metamodel. Recent work with Philippa Conmy, a research assistant in the department, has identified a significant flaw in the implementation, leading to changes to the metamodel. These changes caused existing FPTC models to become inconsistent with the metamodel. Conmy has made

[2]Both have since moved to: `http://www.emn.fr/z-info/atlanmod/index.php/Zoos`

[3]TODO: Ask Richard for grant number. `http://www.modelplex.org/`

available copies of FPTC models from before and after the changes. **Suitability for study:** Suitable for studying co-evolution. Unsuitable for studying synchronisation, because, although the tool includes a transformation, the target models are produced as output from a simulation, never stored and hence cannot become inconsistent with their source model.

xText

xText is an openArchitectureWare (oAW) [openArchitectureWare 2007] tool for generating parsers, metamodels and editors for performing text-to-model transformation. Internally, xText defines a metamodel, which has been changed significantly over the last two years. In several cases, changes have caused inconsistency with existing models. xText provides examples of use, which have been updated alongside the metamodel. **Suitability for study:** Suitable for studying co-evolution. Unsuitable for studying synchronisation.

GMF

The Graphical Modelling Framework (GMF) [Gronback 2009] allows the definition of graphical concrete syntax for metamodels that have been defined in EMF. GMF prescribes a model-driven approach: users of GMF define concrete syntax as a model, which is used to generate a graphical editor. In fact, five models are used together to define a single editor using GMF.

GMF defines the metamodels for graphical, tooling and mapping definition models; and for generator models. The metamodels have changed considerably during the development of GMF. Some changes have caused inconsistency with GMF models. Presently, migration is encoded in Java. Gronback has stated[4] that the migration code is being ported to QVT (a model-to-model transformation language) as the Java code is difficult to maintain.

GMF fulfils almost all of the requirements for the study. Co-evolution data is available, including migration strategies. The GMF source code repository does not contain examples of the kinds of change that cause inconsistency between the models (R8). GMF does, however, have a large number of users, and it might be possible to gather this information from those users. However, contacting GMF users and analysing any data they might supply was not possible in this thesis, due to time constraints. **Suitability for study:** Suitable for studying co-evolution. Unsuitable for studying synchronisation, unless extra data is obtained from GMF users.

Summary of selection

The FPTC and xText projects were selected for a study of co-evolution. No appropriate projects were located for a study of synchronisation. The GMF

[4]Private communication, 2008.

project will not be studied immediately, but reserved for evaluation (Chapter 6).

4.1.3 Other examples

Because only a small number of MDE projects fulfilled all of the requirements, additional data was collected from alternative sources. Firstly, examples were sought from object-oriented systems, which have some similarities to systems developed using MDE. Secondly, examples were discovered during collaboration with colleagues on two projects, both of which involved developing a system using MDE.

Examples of evolution from object-oriented systems

In object-oriented programming, software is constructed by developing groups of related objects. Every object is an instance of (at least) one class. A class is a description of characteristics, which is shared by each of the class's instances (objects). A similar relationship exists between models and metamodels: metamodels comprise meta-classes, which describe the characteristics shared by each of the meta-class's instances (elements of a model). Together, model elements are used to describe one perspective (model) of a system. This similarity between object-oriented programming and metamodelling implied that the evolution of object-oriented systems may be similar to evolution occurring in MDE.

Refactoring is the process of improving the structure of existing code while maintaining its external behaviour. When used as a noun, a refactoring is one such improvement. As discussed in Chapter 3, refactoring of object-oriented systems has been widely studied, perhaps must notably in [Fowler 1999], which provides a catalogue of refactorings for object-oriented systems. For each refactoring, Fowler gives advice and instructions for its application.

To explore their relevance to MDE, the refactorings described in [Fowler 1999] were applied to metamodels. Some were found to be relevant to metamodels, and could potential occur during MDE. Many were found to be irrelevant, belonging to one of the following three categories:

1. **Operational refactorings** focus on restructuring behaviour (method bodies). Most modelling frameworks do not support the specification of behaviour in models.

2. **Navigational refactorings** convert, for example, between bi-directional and uni-directional associations. These changes are non-breaking in EMF, which automatically provides values for the inverse of a reference when required.

3. **Domain-specific refactorings** manage issues specific to object-oriented programming, such as casting, defensive return values, and assertions. These issues are not relevant to metamodelling.

The object-oriented refactorings that can be applied to metamodels provide examples of metamodel evolution. When applied, some of these refactorings potentially cause inconsistency between a metamodel and its models. By using Fowler's description of each refactoring, a migration strategy for updating (co-evolving) inconsistent models was deduced. An example of this process is now presented.

Figure 4.2 illustrates a refactoring that changes a reference object to a value object. Value objects are immutable, and cannot be shared (i.e. any two objects cannot refer to the same value object). By contrast, reference objects are mutable, and can be shared. Figure 4.2 indicates that applying the refactoring restricts the multiplicity of the association (on the Order end) to 1 (implied by the composition); prior to the refactoring the multiplicity is many-valued.

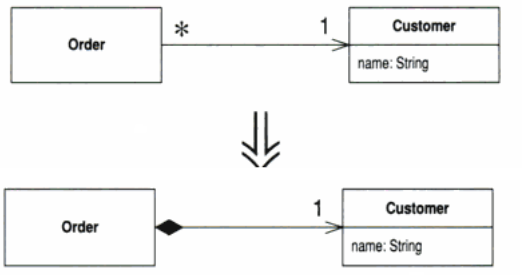


Figure 4.2: Refactoring a reference to a value. Taken from [Fowler 1999][pg183].

Before applying the refactoring, each customer may be associated with more than one order. After the refactoring, each customer should be associated with only one order. Fowler indicates that every customer associated with more than one order should be duplicated, such that one customer object exists for each order. Therefore, the migration strategy in Listing 4.1 is deduced. Using this process, migration strategies were deduced for each of the refactorings that were applicable to metamodelling, and caused inconsistencies between a metamodel and its models.

```
1  for every customer, c
2   for every order, o, associated with c
3    create a new customer, d
4    copy the values of c's attributes into d
5   next o
```

```
6
7   delete c
8  next c
```

Listing 4.1: Migration strategy for the refactoring in pseudo code.

The examples of metamodel evolution based on Fowler's refactorings provided additional data for deriving research requirements. Some parts of the metamodel evolutions from existing MDE projects were later found to be equivalent to Fowler's refactorings, which, to some extent, validates the above claim that evolution from object-oriented systems can be used to reason about metamodel evolution.

However, object-oriented refactorings are used to improve the maintainability of existing systems. In other words, they represent only one of the three reasons for evolutionary change defined by [Sjøberg 1993]. The two other types of change are equally relevant for deriving research requirements, and so object-oriented refactorings alone are not sufficient for reasoning about metamodel evolution.

Research collaborations

As well as the example data located from object-oriented system, collaboration on projects using MDE with two colleagues provided several examples of evolution. A prototypical metamodel to standardise the way in which process-oriented programs are modelled was produced with Adam Sampson, a research assistant at the University of Kent, and an investigation of the feasibility of implementing a tool for generating story-worlds for interactive narratives was conducted with Heather Barber, a postdoctoral researcher in the department.

In both cases, a metamodel was constructed for describing concepts in the domain. The metamodels were developed incrementally and changed over time. The collaborations with Sampson and Barber did not involve constructing model-to-model transformations, but did provide data suitable for a study of co-evolution.

The majority of the changes made in both of these projects relate to changing requirements. In each iteration, existing requirements were refined and new requirements discovered. Neither project required changes to support architectural restructuring. In addition, the work undertaken with Sampson included some changes to adapt the system for use with a different technology than originally anticipated. That is to say, the changes observed represented two of the three reasons for evolutionary change defined by [Sjøberg 1993].

4.1.4 Summary

To summarise, in this section the example data used to analyse existing structures and processes for managing evolution in the context of MDE was discussed. Example data was sought from existing MDE projects, and also from a

related domain and research collaboration. Eight existing MDE projects were located, three of which satisfied the requirements for a study of co-evolutionary changes in the context of model-driven engineering. One of the three projects, GMF, was reserved as a case study and is used for evaluation in Chapter 6. Refactorings of object-oriented programming supplemented the data available from the existing MDE projects. Collaboration with Sampson and Barber yielded further examples of co-evolution.

Due to the lack of examples of model synchronisation, this thesis now focuses on model and metamodel co-evolution.

4.2 Analysing Existing Techniques

The examples of co-evolution identified in the previous section were analysed to discover and compare existing techniques for managing co-evolution. Details of this process are included in Appendix A. This section discusses the results of analysing the examples; namely a deeper understanding of modelling framework characteristics that affect the management of co-evolution (Section 4.2.1) and a categorisation of existing techniques for managing co-evolution (Sections 4.2.2 and 4.2.3) . These results have been published in [Rose *et al.* 2009b, Rose *et al.* 2010e].

4.2.1 Modelling Framework Characteristics Relevant to Co-Evolution

Analysis of the co-evolution examples identified in the previous section highlights characteristics of modern MDE modelling development environments that impact on the way in which co-evolution can be managed.

Model-Metamodel Separation

In modern MDE development environments, *models and metamodels are separated*. Metamodels are developed and distributed to users. Metamodels are installed, configured and combined to form a customised MDE development environment. Metamodel developers have no programmatic access to instance models, which reside in a different workspace and potentially on a different machine. Consequently, metamodel evolution occurs independently to model migration. First, the metamodel is evolved. Subsequently, the users of the metamodel find that their models are out-of-date and migrate their models.

Because of model and metamodel separation, existing techniques for managing co-evolution are either *developer-driven* (the metamodel developer devises an executable migration strategy, which is distributed to the metamodel user with the evolved metamodel) or *user-driven* (the metamodel user devises the migration strategy). In either case, model migration occurs on the machine of the metamodel user, after and independent of metamodel evolution.

Implicit Conformance

Modern MDE development environments *implicitly enforce conformance*. A model is *bound* to its metamodel, typically by constructing a representation in the underlying programming language for each model element and data value. Frequently, binding is strongly-typed: each metamodel type is mapped to a corresponding type in the underlying programming language using mappings defined by the metamodel. Consequently, MDE modelling frameworks do not permit changes to a model that would cause it to no longer conform to its metamodel. Loading a model that does not conforms to its metamodel causes an error. In short, MDE modelling frameworks cannot be used to manage models that do not conform to their metamodel.

Because modelling frameworks can only load models that conform to their metamodel, user-driven migration is always a manual process, in which models are migrated without using the modelling framework. Executable migration strategies can only be used if they are specified with a tool that does not depend on the modelling framework to load the non-conformant models (and, at present, no such tool exists). Typically then, the metamodel user can only perform migration by editing the model directly, normally manipulating its underlying representation (e.g. XMI).

Because modelling frameworks do not permit changes to a model that cause non-conformance, model migration must produce a model that conforms to the evolved metamodel. Therefore, model migration cannot be specified as a combination of co-evolution techniques with each performing some part of the migration, because intermediate steps are not allowed to leave the model in a non-conformant state.

Finally, a further consequence of implicitly enforced conformance is that models cannot be checked for conformance against any metamodel other than their own. Because conformance is always assumed, modern MDE development environments provide limited mechanisms for checking conformance, and typically provide no support for checking conformance to a metamodel other than the one used to construct the model.

4.2.2 User-Driven Co-Evolution

Examples of co-evolution were analysed to discover and compare existing techniques for managing co-evolution. As discussed above, the separation of models and metamodels leads to two processes for co-evolution: *developer-driven* and *user-driven*. No existing research has explored user-driven co-evolution, yet analysis of the co-evolution examples identified in Section 4.1 highlighted several instances of user-driven co-evolution. This section discusses user-driven co-evolution and provides a scenario (based on examples from Section 4.1).

In user-driven co-evolution, the metamodel user performs migration by

loading their models to test conformance, and then rectifying conformance problems by updating non-conformant models. The metamodel developer might guide migration by providing a migration strategy to the metamodel user. Crucially, however, the migration strategy is not executable (e.g. it is written in prose). This is the key distinction between user-driven and developer-driven co-evolution. Only in the latter does the metamodel developer provided an executable model migration strategy.

In some cases, the metamodel user will not be provided with any migration strategy (executable or otherwise) from the metamodel developer. To perform migration, the metamodel user must determine which (if any) model elements no longer conform to the evolved metamodel, and then decide how best to change non-conformant elements to re-establish conformance. This is analogous to developing a legacy system[5].

Users of the same metamodel migrate their models independently. When no migration strategy has been provided by the metamodel developer, each metamodel user must devise their own migration strategy.

Scenario

The following scenario demonstrates user-driven co-evolution. Mark is developing a metamodel. Members of his team, including Heather, install Mark's metamodel and begin constructing models. Mark later identifies new requirements, changes the metamodel, builds a new version of the metamodel, and distributes it to his colleagues.

After several iterations of metamodel updates, Heather tries to load one of her older models, constructed using an earlier version of Mark's metamodel. When loading the older model, the modelling framework reports an error indicating that the model no longer conforms to its metamodel. To load the older model, Heather must reinstall the version of the metamodel to which the older model conforms. But even then, the modelling framework will bind the older model to the old version of the metamodel, and not to the evolved metamodel.

Employing user-driven migration, Heather must trace and repair the loading error directly in the model as it is stored on disk. Model storage formats have typically been optimised to either reduce the size of models on disk or to improve the speed of random access to model elements. Therefore, human usability is not a key requirement for model storage formats (XMI, in particular, is regarded as sub-optimal for use by humans [OMG 2004]) and, consequently, using them for migration is an unproductive and tedious task. When directly editing the underlying format of a model, re-establishing consistency is often a slow, iterative process. For example, EMF uses a multi-pass model parser and hence only reports one category of errors when a model cannot be loaded.

[5]A system for which one has no documentation and whose authors have left the owning organisation. TODO - Fiona suggested reviewing legacy systems in the literature review

After fixing one set of errors, another may be reported. In some cases, models are stored in a binary format and must be changed using a specialised editor, further impeding user-driven co-evolution. For example, models stored using the Connected Data Objects Model Repository (CDO) [Eclipse 2010] are persisted in a relational database, which must be manipulated when non-conformant models are to be edited.

Challenges

The above scenario highlights the two most significant challenges faced when performing user-driven migration. Firstly, the underlying model representation is unlikely to be optimised for human usability. Together with limited support for conformance checking, user-driven migration performed by editing the underlying model representation is error prone and tedious. Secondly, installing a new version of a metamodel plug-in can affect the conformance of models and, moreover, conformance problems are not reported to the user as part of the installation process. These challenges are further elaborated in the Section 4.3, which identifies research requirements.

It is worth noting that the above scenario describes a metamodel with only one user. Metamodels such as those defined in UML, EMF, MOF and GMF have many more users and, hence, user-driven co-evolution is arguably less desirable than developer-driven co-evolution.

4.2.3 Categorisation of Developer-Driven Co-Evolution Techniques

In developer-driven co-evolution, the metamodel developer provides an executable migration strategy along with the evolved metamodel. Model migration might be scheduled automatically by the modelling framework (for example when a model is loaded) or by the metamodel user.

As noted in Section 4.2.2, existing co-evolution research focuses on developer-driven rather than user-driven co-evolution. By applying existing co-evolution approaches to the co-evolution examples identified in Section 4.1, existing developer-driven co-evolution approaches were categorised, compared and contrasted. Three categories of developer-driven co-evolution approach were identified: *manual specification*, *operator-based* and *metamodel matching*. This categorisation has been published in [Rose *et al.* 2009b] and revisited in [Rose *et al.* 2010e]. Each category is now discussed.

Manual Specification

In *manual specification*, the migration strategy is encoded manually by the metamodel developer, typically using a general purpose programming language (e.g. Java) or a model-to-model transformation language (such as QVT [OMG 2005], or ATL [Jouault & Kurtev 2005]). The migration strategy can

manipulate instances of the metamodel in any way permitted by the modelling framework. Manual specification approaches have been used to manage migration in the Eclipse GMF project [Gronback 2009] and the Eclipse MDT UML2 project [Eclipse 2009b]. Compared operator-based and metamodel matching techniques (below), manual specification permits the metamodel developer the most control over model migration.

However, manual specification generally requires the most effort on the part of the metamodel developer for two reasons. Firstly, as well as implementing the migration strategy, the metamodel developer must also produce code for executing the migration strategy. Typically, this involves integration of the migration strategy with the modelling framework (to load and store models) and possibly with development tools (to provide a user interface). Secondly, frequently occurring model migration patterns – such as copying a model element from original to migrated model – are not captured by existing general purpose and model-to-model transformation languages, and so each metamodel developer has to codify these patterns in the chosen migration language.

Operator-based

In *operator-based co-evolution* techniques, a library of *co-evolutionary operators* is provided. Each co-evolutionary operator specifies a metamodel evolution along with a corresponding model migration strategy. For example, the "Make Reference Containment" operator evolves the metamodel such that a non-containment reference becomes a containment reference and migrates models such that the values of the evolved reference are replaced by copies. By composing co-evolutionary operators, metamodel evolution can be performed and a migration strategy can be generated without writing any code. Wachsmuth [Wachsmuth 2007] proposes a library of co-evolutionary operators for MOF metamodels. COPE [Herrmannsdoerfer *et al.* 2009b] is an operator-based co-evolution approach for the Eclipse Modeling Framework.

The usefulness of an operator-based co-evolution approach depends heavily on the richness of the library of co-evolutionary operators that it provides. If the library of co-evolutionary operators cannot be extended, the metamodel developer must use another approach for performing model migration when no appropriate co-evolutionary operator is available. COPE allows metamodel developers to manually specify custom migration strategies when no co-evolutionary operator is appropriate, using a general purpose programming language. (Consequently, custom migration strategies in COPE suffer one of the same limitations as manual specification approaches: model migration patterns are not captured in the language used to specify migration strategies). A custom migration strategy can be imported as a co-evolutionary operator if it can be specified independently of its metamodel. Importing an operator in COPE requires the metamodel developer to rewrite their custom

migration strategy, removing references to their metamodel.

As using co-evolutionary operators to express migration require the metamodel developer to write no code, it seems that operator-based co-evolution approaches should seek to provide a large library of co-evolutionary operators, so that at least one operator is appropriate for every co-evolution that a metamodel developer may wish to perform. However, as discussed in [Lerner 2000], a large library of operators increases the complexity of specifying migration. To demonstrate, Lerner considers moving a feature from one type to another. This could be expressed by sequential application of two operators called, for example, `delete_feature` and `add_feature`. However, the semantics of a `delete_feature` operator are likely to dictate that the values of that feature will be removed during migration and hence, `delete_feature` is unsuitable when specifying that a feature has been moved. To solve this problem, a `move_feature` operator could be introduced, but then the metamodel developer must understand the difference between the two ways in which moving a type can be achieved, and carefully select the correct one. Lerner provides other examples which further elucidate this issue (such as introducing a new type by splitting an existing type). As the size of the library of co-evolutionary operators grows, so does the complexity of selecting appropriate operators and, hence, the complexity of performing metamodel evolution.

Clear communication of the effects of each co-evolutionary operator (on both the metamodel and its instance models) can improve the navigability of large libraries of co-evolutionary operators. COPE, for example, provides a name, description, list of parameters and applicability constraints for each co-evolutionary operator. An example, taken from COPE's library[6], is shown below.

> **Make Reference Containment**
>
> In the metamodel, a reference is made [into a] containment. In the model, its values are replaced by copies.
>
> *Parameters*:
>
> - `reference`: The reference
>
> *Constraints*:
>
> - The `reference` must not already be containment.

COPE operators are defined in Groovy[7]. To select the correct operator, users can read descriptions (an example is shown above), examine the source code, or try executing the operator (an undo command is provided). There

[6] `http://cope.in.tum.de/pmwiki.php?n=Operations.MakeContainment`

[7] A dynamic, strongly typed object-oriented programming language for the Java platform

are no formally defined semantics for COPE's operators, and verifying their effects involves inspecting the model on which they are executed.

Other techniques can be used to try to improve the navigability of large libraries of co-evolutionary operators. COPE, for example, restricts the choice of operators to only those that can be applied to the currently selected metamodel element. Finding a balance between richness and navigability is a key challenge in defining libraries of co-evolutionary operators for operation-based co-evolution approaches. Analogously, a known challenge in the design of software interfaces is the trade-off between a rich and a concise interface [Bloch 2005].

To perform metamodel evolution using co-evolutionary operators, the library of co-evolutionary operators must be integrated with tools for editing metamodels. COPE, for instance, provides integration with the EMF tree-based metamodel editor. However, some developers edit their metamodels using a textual syntax, such as Emfatic [IBM 2005]. In general, freeform text editing is less restrictive than tree-based editing (because in the latter, the metamodel is always structurally sound whereas in the former, the text does not always have to compile). Consequently, it is not clear whether operator-based co-evolution can be used with all categories of metamodel editing tool.

Metamodel Matching

In *metamodel matching*, a migration strategy is inferred by analysing the evolved metamodel and the *metamodel history*. Metamodel matching approaches use one of two categories of metamodel history; either the original metamodel (*differencing* approaches) or the changes made to the original metamodel to produce the evolved metamodel (*change recording* approaches). The analysis of the evolved metamodel and the metamodel history yields a *difference model* [Cicchetti *et al.* 2008], a representation of the changes between original and evolved metamodel. The difference model is used to infer a migration strategy, typically by using a higher-order model-to-model transformation[8] to produce a model-to-model transformation from the difference model. Cicchetti et al. [Cicchetti *et al.* 2008] and Garcés et al. [Garcés *et al.* 2009] describe metamodel matching approaches. More specifically, both describe differencing approaches. There exist no pure change recording approaches, although COPE [Herrmannsdoerfer *et al.* 2009b] uses change recording to support the specification of custom model migration strategies.

Compared to manual specification and operator-based co-evolution, metamodel matching requires the least amount of effort from the metamodel developer who needs only to evolve the metamodel and provide a metamodel history. However, for some types of metamodel change, there is more than one feasible model migration strategy. For example, when a metaclass is deleted,

[8] A model-to-model transformation that consumes or produces a model-to-model transformation is termed a higher-order model transformation.

one feasible migration strategy is to delete all instances of the deleted metaclass. Alternatively, the type of each instance of the deleted metaclass could be changed to another metaclass that specifies equivalent structural features.

To select the most appropriate migration strategy from all feasible alternatives, a metamodel matching approach often requires guidance, because the metamodel changes alone do not provide enough information to correctly distinguish between feasible migration strategies. Existing metamodel matching approaches use heuristics to determine the most appropriate migration strategy. These heuristics sometimes lead to the selection of the wrong migration strategy.

Because metamodel matching approaches use heuristics to select a migration strategy, it can sometimes be difficult to reason about which migration strategy will be selected. For domains where predictability, completeness and correctness are a primary concern (e.g. safety critical or security critical systems, or systems that must undergo certification with respect to a relevant standard), such approaches are unsuitable, and deterministic approaches that can be demonstrated to produce correct, predictable results will be required.

Before discussing the benefits and limitations of differencing and change recording metamodel matching approaches, an example of co-evolution is introduced.

Example The following example was observed during the development of the Epsilon FPTC tool (summarised in Section 4.1 and described in [Paige *et al.* 2009]). The source code is available from EpsilonLabs[9]. Figure 4.3 illustrates the original metamodel in which a `System` comprises any number of `Blocks`. A `Block` has a name, and any number of `successor Blocks`; `predecessors` is the inverse of the `successors` reference.

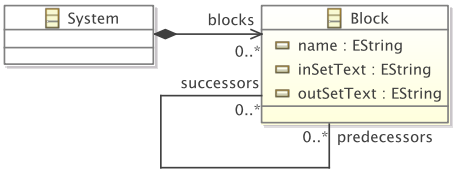


Figure 4.3: Original metamodel, prior to evolution [Rose *et al.* 2009b].

Further analysis of the domain revealed that extra information about the relationship between `Blocks` needed to be stored. The evolved metamodel is shown in Figure 4.4. The `Connection` class is introduced to capture this extra information. `Blocks` are no longer related directly to `Blocks`, instead they are related via an instance of the `Connection` class. The

[9] `http://sourceforge.net/projects/epsilonlabs/`

`incomingConnections` and `outgoingConnections` references of `Block` are used to relate `Blocks` to each other via an instance of `Connection`.

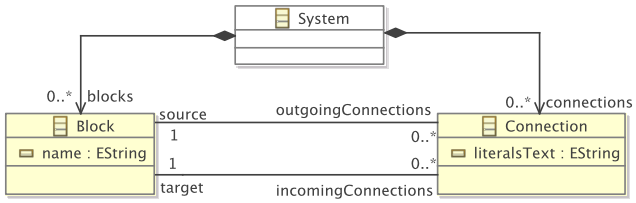


Figure 4.4: Evolved metamodel with Connection metaclass [Rose *et al.* 2009b].

A model that conforms to the original metamodel (Figure 4.3) might not conform to the evolved metamodel (Figure 4.4). Below is a description of the strategy used by the Epsilon FPTC tool to migrate a model from original to evolved metamodel and is taken from [Rose *et al.* 2009b]:

1. For every instance, `b`, of `Block`:

   For every successor, `s`, of `b`:

   Create a new instance, `c`, of `Connection`.

   Set `b` as the `source` of `c`.

   Set `s` as the `target` of `c`.

   Add `c` to the `connections` reference of the `System` containing `b`.

2. And nothing else changes.

Differencing and change recording metamodel matching approaches are now compared and contrasted.

Change recording In change recording approaches, metamodel evolution is monitored by a tool, which records a list of primitive changes (e.g. Add class named `Connection`, or Change the type of feature `successors` from `Block` to `Connection`). The record of changes may be reduced to a normal form, which might remove redundancy, but might also erase useful information. In change recording, some types of metamodel evolution can be more easily recognised than with differencing. With change recording, renaming of a metamodel element from X to Y can be distinguished from the following sequence: remove a metamodel element called X, add a metamodel element called Y. With differencing, this distinction is not possible.

In general, more than one combination of primitive changes can be used to achieve the same metamodel evolution. However, when recording changes, the way in which a metamodel is evolved affects the inference of migration strategy. In the example presented above, the `outgoingConnections` reference (shown in Figure 4.4) could have been produced by changing the name and type of the `successors` reference (shown in Figure 4.3). In this case, the record of changes would indicate that the new `outgoingConnections` reference is an evolution of the `successors` reference, and consequently an inferred migration strategy would be likely to migrate values of `successors` to values of `outgoingConnections`. Alternatively, the metamodel developer may have elected to delete the `successors` reference and then create the `outgoingConnections` reference afresh. In this record of changes, it is less obvious that the migration strategy should attempt to migrate values of `successors` to values of `outgoingConnections`. Change recording approaches require the metamodel developer to consider the way in which their metamodel changes will be interpreted.

Change recording approaches require facilities for monitoring metamodel changes from the metamodel editing tool, and from the underlying modelling framework. As with operation-based co-evolution, it is not clear to what extent change recording can be supported when a textual syntax is used to evolve a metamodel. A further challenge is that the granularity of the metamodel changes that can be monitored influences the inference of the migration strategy, but this granularity is likely to be controlled by and specific to the implementation of the metamodelling language. In his thesis, [Sprinkle 2008] discusses this issue, devising a normal form, optimised for his approach, to which a record of changes can be reduced.

Differencing In differencing approaches, the original and evolved metamodels are compared to produce the difference model. Unlike change recording, metamodel evolution may be performed using any metamodel editor; there is no need to monitor the primitive changes made to perform the metamodel evolution. However, as discussed above, not recording the primitive changes can cause some categories of change to become indistinguishable, such as renaming versus a deletion followed by an addition.

To illustrate this problem further, consider again the metamodel evolution described above. A comparison of the original (Figure 4.3) and evolved (Figure 4.4) metamodels shows that the references named `successors` and `predecessors` no longer exist on `Block`. However, two other references, named `outgoingConnections` and `incomingConnections`, are now present on `Block`. A differencing approach might deduce (correctly, in this case) that the two new references are evolutions of the old references. However, no differencing approach is able to determine which mapping is correct from the following two possibilities:

- `successors` evolved to `incomingConnections`, and `predecessors` evolved to `outgoingConnections`.
- `successors` evolved to `outgoingConnections`, and `predecessors` evolved to `incomingConnections`.

The choice between these two possibilities can only made by the metamodel developer, who knows that `successors` (`predecessors`) is semantically equivalent to `outgoingConnections` (`incomingConnections`). As shown by this example, fully automatic differencing approaches cannot always infer a migration strategy that will capture the semantics desired by the metamodel developer.

4.2.4 Summary

Analysis of existing co-evolution techniques has led to a deeper understanding of modelling frameworks characteristics that are relevant for co-evolution, to the discovery of user-driven, rather than developer-driven, model migration and to a categorisation of co-evolution techniques.

Modern MDE modelling frameworks separate models and metamodels and, hence, co-evolution is a two-step process. To facilitate model migration, metamodel developers may codify an executable migration strategy and distribute it along with the evolved metamodel (developer-driven migration). Because modelling frameworks implicitly enforce conformance, the underlying representation of non-conformant models is edited by the metamodel user when no executable migration strategy is provided by the metamodel developer (user-driven migration).

User-driven migration, which has not yet been explored in existing research, was observed in several of the co-evolution examples discussed in Section 4.1. In situations where the metamodel developer has not specified or cannot specify an executable migration strategy, user-driven migration is required.

In Section 4.1, existing techniques for performing developer-driven co-evolution were compared, contrasted and categorised. The categorisation highlights a trade-off between flexibility and effort for the metamodel-developer when choosing between categories of approach. Manual specification affords the metamodel developer more flexibility in the specification of the migration strategy, but, because languages that do not capture re-occurring model migration patterns are typically used, may require more effort. By contrast, metamodel matching approaches seek to infer a migration strategy from a metamodel history and hence require less effort from the metamodel developer. However, a metamodel matching approach affords the metamodel developer less flexibility, and may restrict the metamodel evolution process. (For example, the order or way in which the metamodel is changed may influence the inference of the migration strategy). Operator-based approaches occupy

the middle-ground: by restricting the way in which metamodel evolution is expressed, an operator-based approach can be used to infer a migration strategy. The metamodel developer selects appropriate operators that express both metamodel evolution and model migration. Operator-based approaches require a specialised metamodel editor, and it is not yet clear whether they can be applied when a metamodel is represented with a freeform (e.g. textual) rather than a structured (e.g. tree-based) syntax.

4.3 Requirements Identification

In Chapter 3, research objectives were identified. Based on the analysis presented in this chapter, the objectives were refined, deriving requirements for this thesis. [10]

Below, the thesis requirements are presented in three parts. The first identifies requirements that seek to extend and enhance support for managing model and metamodel co-evolution with modelling frameworks. The second summarises and identifies requirements for enhancing the user-driven co-evolution process discussed in Section 4.2.2. Finally, the third identifies requirements that seek to improve the spectrum of existing developer-driven co-evolution techniques.

4.3.1 Explicit conformance checking

Section 4.2.1 discussed characteristics of modelling frameworks relevant to managing co-evolution. Because modelling frameworks typically enforce model and metamodel conformance implicitly, they cannot be used to load non-conformant models. Consequently, user-driven co-evolution techniques are restricted to processes which involve editing a model in its storage representation. Because human usability is not normally a key requirement for model storage representations, implicitly enforcing conformance causes challenges for user-driven co-evolution. Furthermore, modelling frameworks that implicitly enforce conformance understandably provide little support for explicitly checking the conformance of a model with other metamodels (or other versions of the same metamodel). As discussed in Section 4.2.1, explicit conformance checking is useful for determining whether a model needs to be migrated (during the installation of a newer version of its metamodel, for example).

Therefore, the following requirement was derived: *This thesis must investigate the extension of existing modelling frameworks to support the loading of non-conformant models and conformance checking of models against other metamodels.*

[10] TODO: Add a linking paragraph that describes high-level requirements and relates them to the following subsections.

4.3.2 User-driven co-evolution

When a metamodel change will affect conformance in only a small number of models, a metamodel developer may decide that the extra effort required to specify an executable migration strategy is too great, and prefer a user-driven co-evolution technique. Section 4.2.2 introduced user-driven co-evolution techniques, highlighting several of the challenges faced when they are applied. Those challenges are now summarised and used to derive requirements for this thesis.

Because modelling frameworks typically cannot be used to load non-conformant models, user-driven co-evolution involves editing the storage representation of a model. As discussed above, model storage representations are typically not optimised for human use and hence user-driven co-evolution can be error-prone and time consuming. When a multi-pass parser is used to load models (as is the case with EMF), user-driven co-evolution is an iterative process, because not all conformance errors are reported at once.

Therefore, the following requirement was derived: *This thesis must demonstrate a user-driven co-evolution process that enables the editing of non-conformant models without directly manipulating the underlying storage representation and provides a conformance report for the original model and evolved metamodel.*

4.3.3 Developer-driven co-evolution

The comparison of developer-driven co-evolution techniques (Section 4.2.3) highlights variation in the languages used for codifying model migration strategies. Java, Groovy and ATL were among those used. More specifically, the model migration strategy languages varied in their scope (general-purpose programming languages vs model transformation languages) and category of type system. Furthermore, the amount of processing performed when executing a model migration strategy also varied: some techniques only load a model, execute the model migration strategy using an existing execution engine and store the model, while others perform significant processing in addition to the computation specified in the model migration strategy. COPE, for example, transforms models to a metamodel-independent representation before migration is executed, and back to a metamodel-specific representation afterwards.

Of the three categories of developer-driven co-evolution technique identified in Section 4.2.3, only manual specification (in which the metamodel developer specifies the migration strategy by hand) always requires the use of a migration strategy language. Nevertheless, both operator-based and metamodel matching approaches might utilise a migration strategy language in particular circumstances. Some operator-based approaches, such as COPE, permit manual specification of a model migration strategy when no co-evolutionary operator is appropriate. For describing the effects of co-evolutionary operators, the model migration part of an operator could be described using a model

migration strategy language. When application of a metamodel matching approach leads to the inference of more than one feasible migration strategy, the metamodel developer could choose between alternatives that are presented in a migration strategy language. To some extent then, the choice of model migration strategy language influences the effectiveness of a developer-driven co-evolution technique.

Given the variations in existing model migration strategy languages and the influence of those languages on existing developer-driven co-evolution techniques, the following requirement was derived: *This thesis must compare and evaluate existing languages for specifying model migration strategies.*

As discussed in Section 4.2.3, existing manual specification techniques do not provide model migration strategy languages that capture patterns specific to model migration. Developers must re-invent solutions to commonly occurring model migration patterns, such as copying an element from the original to the migrated model. In some cases, manual specification techniques require the developer to implement, in addition to a migration strategy, infrastructure features for loading and storing models and for interfacing with the metamodel user.

A domain-specific language (discussed in Chapter 3) provides one way to capture re-occurring patterns. When accompanied with an execution engine that encapsulates infrastructure features, a domain-specific language is a common way for specifying model management operations in modern model-driven development environments. Domain-specific languages are provided by model-to-model (M2M) transformation tools such as ATL [ATLAS 2007], VIATRA [Varró & Balogh 2007], workflow architectures such as oAW [openArchitectureWare 2007], and model-to-text (M2T) transformation tools such as MOFScript [Oldevik *et al.* 2005] and XPand [openArchitectureWare 2007].

Given the apparent appropriateness of a domain-specific language for specifying model migration and that no language has yet been devised, the following requirement was derived: *This thesis must implement and evaluate a domain-specific language for specifying and executing model migration strategies, comparing it to existing languages for specifying model migration strategies.*

4.4 Chapter Summary

To be completed.

Chapter 5

Implementation

Section 4.3 identified requirements for structures and processes for managing co-evolution. In this chapter, the way in which this thesis approaches those requirements is described. Several related solutions were implemented, using domain-specific languages, automation and extensions to existing modelling technologies. Figure 5.1 summarises the structure of the chapter. To better support co-evolution and to overcome restrictions with existing modelling frameworks, a metamodel-independent syntax was devised and implemented, enabling model and metamodel decoupling and consistency checking (Section 5.1). To address some of the challenges faced in user-driven co-evolution, an OMG specification for an alternative, textual modelling notation was implemented (Section 5.2). Model migration languages were identified, analysed and compared, leading to the derivation and implementation of a new model transformation language tailored for model migration and centred around a novel approach to relating source and target model elements (Section 5.4).

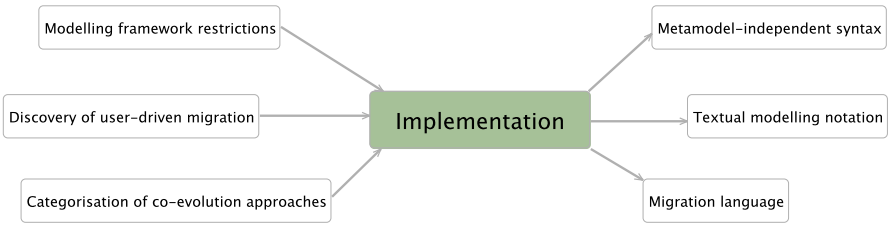


Figure 5.1: Implementation chapter overview.

5.1 Metamodel-Independent Syntax

Section 4.2.1 discussed the way in which modelling frameworks implicitly enforce conformance. Because of this, modelling frameworks cannot be used

to load non-conformant models, and provide little support for checking the conformance of a model with other metamodels or other versions of a metamodel. In Section 4.3, these concerns lead to the identification of the following requirement: *This thesis must investigate the extension of existing modelling frameworks to support the loading of non-conformant models and conformance checking of models against other metamodels.*

This section describes the way in which existing modelling frameworks load and store models using a metamodel-specific syntax. An alternative storage representation is motivated by highlighting the problems that a metamodel-specific syntax poses for managing and automating co-evolution. The way in which automatic consistency checking can be performed using the alternative storage representation is demonstrated. The work presented in this section has been published in [Rose *et al.* 2009a].

5.1.1 Model Storage Representation

Throughout a model-driven development process, modelling frameworks are used to load and store models. XML Metadata Interchange (XMI) [OMG 2007c], the OMG standard for exchanging MOF-based models, is the canonical model representation used by many contemporary modelling frameworks. XMI specifies the way in which models should be represented in XML.

An XMI document defines one or more namespaces from which type information is drawn. For example, XMI itself provides a namespace for specifying the version of XMI being used. Metamodels are referenced via namespaces, allowing the specification of elements that instantiate metamodel types.

As discussed in Section 4.2.1, modelling frameworks bind a model to its metamodel using the underlying programming language. The metamodel defines the way in which model elements will be bound, and frequently, binding is strongly-typed: each metamodel type is mapped to a corresponding type in the underlying programming language.

Listing 5.1 shows XMI for an exemplar model conforming to a metamodel that defines `Person` as a metaclass with three features: a string-valued `name`, an optional reference to a `Person`, `mother`, and another optional reference to a `Person`, `father`.

```
1 <?xml version="1.0" encoding="ASCII"?>
2 <xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
      xmlns:families="http://www.cs.york.ac.uk/families">
3   <families:Person xmi:id="_xNSb8KfZEd,0dNl1iq3EdQ" name="Franz" mother=
        "_6ef33ff010b31df8a39080" father="_F520cDaa0jN,i10s8xZp2a" />
4   <families:Person xmi:id="_6ef33ff010b31df8a39080" name="Julie" />
5   <families:Person xmi:id="_F520cDaa0jN,i10s8xZp2a" name="Hermann" />
6 </xmi:XMI>
```

Listing 5.1: Exemplar person model in XMI

The model shown in Listing 5.1 contains three `Persons`, Franz, Julie and Hermann. Julie is the mother and Hermann is the father of Franz. The mothers and fathers of Julie and Hermann are not specified. On line 2, the XMI document specifies that the families namespace will be used to refer to types defined by the metamodel with the identifier: `http://www.cs.york.ac.uk/families`. Each person defines an XMI ID (a universally unique identifier), and a name. The IDs are used for inter-element references, such as for the values of the mother and father features.

Binding a model element involves instantiating, in the underlying programming language, the metamodel type, and populating the attributes of the instantiated object with values that correspond to those specified in the model. Because an XMI document refers to metamodel types and features by name, binding fails when a model does not conform to its metamodel.

5.1.2 Binding to a generic metamodel

For situations when a model does not conform to its metamodel, this thesis proposes an alternative deserialisation mechanism, which binds a model to a *generic* metamodel. A generic metamodel reflects the characteristics of the metamodelling language and consequently every model conforms to the generic metamodel. Figure 5.2 shows a minimal version of a generic metamodel for MOF. Model elements are bound to `Object`, data values to `Slot`.

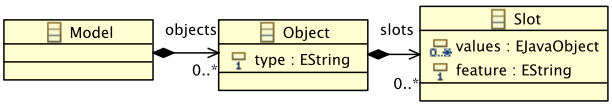


Figure 5.2: A generic metamodel.

Using the metamodel in Figure 5.2 in conjunction with MOF, conformance constraints can be expressed, as shown below. A minimal subset of MOF is shown in Figure 5.3.

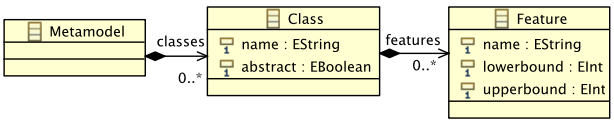


Figure 5.3: Minimal MOF metamodel.

The following constraints between metamodels (e.g. instances of MOF, Figure 5.3) and models represented with a generic metamodel (e.g. instances of Figure 5.2) can be used to express conformance:

1. Each object's type must be the name of some non-abstract metamodel class.

2. Each object must specify a slot for each mandatory feature of its type.

3. Each slot's feature must be the name of a metamodel feature. That metamodel feature must belong to the slot's owning object's type.

4. Each slot must be multiplicity-compatible with its feature. More specifically, each slot must contain at least as many values as its feature's lower bound, and at most as many values as its feature's upper bound.

5. Each slot must be type-compatible with its feature.

The way in which type-compatibility is checked depends on the way in which the modelling framework is implemented, and on its underlying programming language. EMF, for example, is implemented in Java and exposes some services for checking the type compatibility of model data with metamodel features. All metamodel features are typed and their types provide methods for determining the underlying programming language representation. Type compatibility checks can be implemented using these methods.

Conformance constraints vary over modelling languages. For example, Ecore, the modelling language of EMF, is similar to but not the same as MOF. For example, metamodel features defined in Ecore can be marked as transient (not stored to disk) and unchangeable (read-only). In EMF, extra conformance constraints are required which restrict the feature value of slots to only non-transient, changeable features.

5.1.3 Example

By binding a model not to the underlying programming languages types defined in its metamodel but to the generic metamodel presented in Figure 5.2, conformance can be checked using the above constraints. Binding the exemplar XMI in Listing 5.1 to the generic metamodel shown in Figure 5.2 produces three Objects, all with type "Person". Each class object contains a slot whose feature is name, one with the value "Franz", one with the value "Julie" and the other with the value "Hermann". The object containing the slot with value "Franz" contains two further slots: one whose feature is mother and whose value is a reference[1] to the object that contains the name slot with the value "Julie" and one whose feature is father and whose value is a reference to the object containing "Hermann". A UML object diagram for this instantiation of the generic metamodel is shown in Figure 5.4. Instances of object (slot) are shaded grey (white).

[1]The generic metamodel used in this thesis implements reference values using the proxy design pattern [Gamma *et al.* 1995].

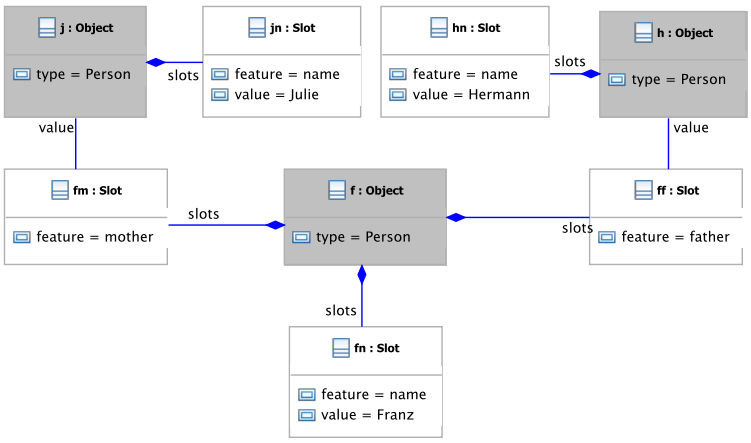


Figure 5.4: Exemplar instantiation of generic metamodel.

After binding to the generic metamodel, the conformance of a model can be checked against any metamodel. Suppose the metamodel used to construct the XMI shown in Figure 5.2 has now evolved. The mother and father references have been removed, and replaced by a unifying parents reference. Conformance checking for the object representing Franz will fail because it defines slots for features "mother" and "father", which are no longer defined for the metamodel class "Person". More specifically, the model element representing Franz does not satisfy conformance constraint 4 from Section 5.1.2, which states that: each slot's feature must be the name of a metamodel feature. That metamodel feature must belong to the slot's owning object's type.

5.1.4 Applications

As this section has shown, binding to a metamodel independent syntax is an alternative model deserialisation mechanism that can be used when a model no longer conforms to its metamodel and to check the conformance of a model with any metamodel. The metamodel independent syntax described in this section is used throughout this chapter to support other structures and processes for co-evolution.

In Section 5.2, a textual modelling notation is integrated with the metamodel independent model representation discussed here. In Section 5.4, a domain-specific language for migration uses metamodel independent syntax to perform partial migration by producing models that conform to a generic metamodel rather than their evolved metamodel.

One of the model migration tools discussed in Section 5.3, COPE [Herrmannsdoerfer *et al.* 2009b], proposes a model migration language that manipulates models via a metamodel-independent syntax. Some of the strengths and weaknesses of that approach

and using a metamodel-independent syntax in that context are described in Sections 5.3.2 and 5.3.3.

Automatic Consistency Checking

In addition to the applications outlined above, a metamodel independent syntax is particularly useful during metamodel installation. As discussed in Section 4.2.1, metamodel developers do not have access to downstream models. Consequently, instances of a metamodel may become non-conformant after a new version of a metamodel plug-in is installed. By default, an EMF metamodel plug-in does not check conformance during plug-in installation and non-conformant models are only detected when the user attempts to load them.

To enable conformance checking as part of metamodel installation in EMF, the binding to a generic metamodel discussed above has been integrated with Concordance [Rose *et al.* 2010b] in joint work with Dimitrios S. Kolovos, a lecturer in this department, Nicholas Drivalos, a research associate in this department and James R. Williams, a research student in this department.

Concordance provides a light-weight and efficient mechanism for resolving inter-model references, including the references between models and their metamodels. Concordance can be used to efficiently determine the instances of a metamodel, which is otherwise only possible with a brute force search of a development workspace.

The integration work involved extending Concordance such that, after the installation of a metamodel plug-in, models that conform to any previous version of the metamodel are identified. Those models are checked for conformance with the new metamodel. As such, conformance checking occurs automatically and during metamodel installation. Conformance problems are detected and reported immediately, rather than when the user next attempts to load an affected model. By integrating conformance checking with Concordance, improved scalability is achieved, as demonstrated in [Rose *et al.* 2010b].

5.2 Textual Modelling Notation

The analysis of co-evolution examples in Chapter 4 highlighted two categories of process for managing co-evolution, developer-driven and user-driven. In the former, migration strategies are executable, while in the latter they are not. Performing user-driven co-evolution with modelling frameworks presents two key challenges that have not been explored by existing research. Firstly, user-driven co-evolution frequently involves editing the storage representation of the model, such as XMI. Model storage representations are typically not optimised for human use and hence user-driven co-evolution can be error-prone. Secondly, non-conformant model elements must be identified during

user-driven co-evolution. When a multi-pass parser is used to load models, as is the case with EMF, not all conformance problems are reported at once, and user-driven co-evolution is an iterative process. In Section 4.3, these challenges lead to the identification of the following requirement: *This thesis must demonstrate a user-driven co-evolution process that enables the editing of non-conformant models without directly manipulating the underlying storage representation and provides a sound and complete conformance report for the original model and evolved metamodel.*

The remainder of this section describes a textual notation for models, which has been implemented for EMF, and discusses the way in which the notation has been integrated with the metamodel independent syntax described in Section 5.1 to produce conformance reports.

5.2.1 Human-Usable Textual Notation

The OMG's Human-Usable Textual Notation (HUTN) [OMG 2004] defines a textual modelling notation, which aims to conform to human-usability criteria [OMG 2004]. There is no current reference implementation of HUTN: the Distributed Systems Technology Centre's TokTok project (an implementation of the HUTN specification) is inactive (and the source code can no longer be found), whilst work on implementing the HUTN specification by Muller and Hassenforder [Muller & Hassenforder 2005] has been abandoned in favour of Sintaks [IRISA 2007], which operates on domain-specific concrete syntax.

Model storage representations are often optimised to reduce storage space or to increase the speed of random access, rather than for human usability. By contrast, the HUTN specification states its primary design goal as human-usability and "this is achieved through consideration of the successes and failures of common programming languages" [OMG 2004, Section 2.2]. The HUTN specification refers to two studies of programming language usability to justify design decisions. Because no reference implementation exists, the specification does not evaluate the human-usability of the notation. This thesis proposes that HUTN be used instead of XMI for user-driven co-evolution. Further discussion of the human-usability of HUTN is deferred to Chapter 6.

Like the generic metamodel presented in Section 5.1, HUTN is a metamodel-independent syntax for MOF. In this section, the core syntax and key features of HUTN are introduced. The complete definition is available in [OMG 2004]. To illustrate usage of the notation, the MOF-based metamodel of families in Figure 5.5 is used. (A nuclear family "consists only of a father, a mother, and children." [Merriam-Webster 2010]).

Basic Notation

Listing 5.2 shows the construction of an *object* in HUTN, here an instance of the Family class from Figure 5.5. Line 1 specifies the package containing the

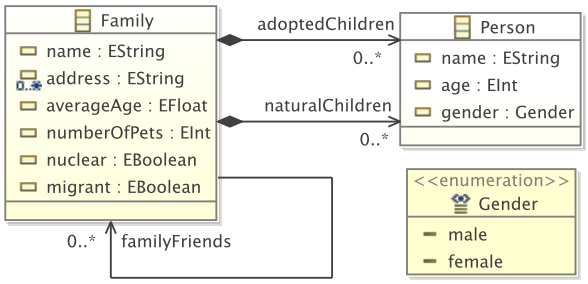


Figure 5.5: Exemplar families metamodel. (Shading is irrelevant).

classes to be constructed (`FamilyPackage`) and a corresponding identifier (`families`), used for fully-qualifying references to objects (Section 5.2.1). Line 2 names the class (`Family`) and gives an identifier for the object (`The Smiths`). Lines 3 to 7 define *attribute values*; in each case, the data value is assigned to the attribute with the specified name. The encoding of the value depends on its type: strings are delimited by any form of quotation mark; multi-valued attributes use comma separators, etc.

The metamodel in Figure 5.5 defines a *simple reference* (familyFriends) and two *containment references* (adoptedChildren; naturalChildren). The HUTN representation embeds a contained object directly in the parent object, as shown in Listing 5.3. A simple reference can be specified using the type and identifier of the referred object, as shown in Listing 5.4. Like attribute values, both styles of reference are preceded by the name of the meta-feature.

```
1 FamilyPackage "families" {
2    Family "The Smiths" {
3       nuclear: true
4       name: "The Smiths"
5       averageAge: 25.7
6       numberOfPets: 2
7       address: "120 Main Street", "37 University Road"
8    }
9 }
```

Listing 5.2: Specifying attributes with HUTN.

```
1 FamilyPackage "families" {
2    Family "The Smiths" {
3       naturalChildren: Person "John" { name: "John" },
4                        Person "Jo" { gender: female }
5    }
```

```
6 }
```

Listing 5.3: Instantiation of naturalChildren – a HUTN containment reference.

```
1 FamilyPackage "families" {
2    Family "The Smiths" {
3       familyFriends: Family "The Does"
4    }
5    Family "The Does" {}
6 }
```

Listing 5.4: Specifying a simple reference with HUTN.

Keywords and Adjectives

While HUTN is unlikely to be as concise as a metamodel-specific concrete syntax, the notation does define syntactic shortcuts to make model specifications more compact. Shortcut use is optional, and the HUTN specification aims to make their syntax intuitive [OMG 2004, pg2-4]. Two example notational shortcuts are described here, to illustrate some of the ways in which HUTN can be used to construct models in a concise manner.

When specifying a *Boolean-valued attribute*, it is sufficient to simply use the attribute name (value `true`), or the attribute name prefixed with a tilde (value `false`). When used in the body of the object, this style of Boolean-valued attribute represents a *keyword*. A keyword used to prefix an object declaration is called an *adjective*. Listing 5.5 shows the use of both an attribute keyword (`~nuclear` on line 6) and adjective (`~migrant` on line 2).

```
1 FamilyPackage "families" {
2    ~migrant Family "The Smiths" {}
3
4    Family "The Does" {
5       averageAge: 20.1
6       ~nuclear
7       name: "The Does"
8    }
9 }
```

Listing 5.5: Using keywords and adjectives in HUTN.

Inter-Package References

To conclude the summary of the notation, two advanced features defined in the HUTN specification are discussed. The first enables objects to refer to other objects in a different package, while the second provides means for specifying the values of a reference for all objects in a single construct (which can be used, in some cases, to simplify the specification of complicated relationships).

```
1  FamilyPackage "families" {
2     Family "The Smiths" {}
3  }
4  VehiclePackage "vehicles" {
5     Vehicle "The Smiths' Car" {
6        owner: FamilyPackage.Family "families"."The Smiths"
7     }
8  }
```

Listing 5.6: Referencing objects in other packages with HUTN.

To reference objects between separate package instances in the same document, the package identifier is used to construct a fully-qualified name. Suppose a second package is introduced to the metamodel in Figure 5.5. Among other concepts, this package introduces a Vehicle class, which defines an owner reference of type Family. Listing 5.6 illustrates the way in which the owner feature can be populated. Note that the fully-qualified form of the class utilises the names of elements of the metamodel, while the fully-qualified form of the object utilises only HUTN identifiers defined in the current document.

The HUTN specification defines name scope optimisation rules, which allow the definition above to be simplified to: `owner: Family "The Smiths"`, assuming that the VehiclePackage does not define a Family class, and that the identifier "The Smiths" is not used in the VehiclePackage block, or this HUTN document is configured to require unique identifiers over the entire document.

Alternative Reference Syntax

In addition to the syntax defined in Listings 5.3 and 5.4, the value of references may be specified independently of the object definitions. For example, Listing 5.7 demonstrates this alternate syntax by defining The Does as friends with both The Smiths and The Bloggs.

```
1  FamilyPackage "families" {
2     Family "The Smiths" {}
3     Family "The Does" {}
4     Family "The Bloggs" {}
5
6     familyFriends {
7        "The Does" "The Smiths"
8        "The Does" "The Bloggs"
9     }
10 }
```

Listing 5.7: Using a reference block in HUTN.

Listing 5.8 illustrates a further alternative syntax for references, which employs an infix notation.

```
1 FamilyPackage "families" {
2     Family "The Smiths" {}
3     Family "The Does" {}
4     Family "The Bloggs" {}
5
6     Family "The Smiths" familyFriends Family "The Does";
7     Family "The Smiths" familyFriends Family "The Bloggs";
8 }
```

Listing 5.8: Using an infix reference in HUTN.

The reference block (Listing 5.7) and infix (Listing 5.8) notations are syntactic variations on – and have identical semantics to – the reference notation shown in Listings 5.3 and 5.4.

Customisation via Configuration

Some limited customisation of HUTN for particular metamodels can be achieved using *configuration files*. Customisations permitted include a parametric form of object instantiation (not yet implemented); renaming of metamodel elements; giving default values for attributes; and stating an attribute whose values are used to infer a default identifier.

5.2.2 Epsilon HUTN

To investigate the extent to which HUTN can be used during user-driven co-evolution, an implementation, Epsilon HUTN, was constructed. This section describes the way in which Epsilon HUTN was implemented using a combination of model-management operations. From text conforming to the HUTN syntax (described above), Epsilon HUTN produces an equivalent model that can be managed with the Eclipse Modeling Framework [Steinberg *et al.* 2008]. The sequel demonstrates the way in which Epsilon HUTN can be used for user-driven co-evolution.

Implementation of Epsilon HUTN

Epsilon HUTN, makes extensive use of the Epsilon model management platform, which was introduced in Section 2.3.2. Epsilon provides infrastructure for implementing uniform and interoperable model management languages, for performing tasks such as model merging, model transformation and inter-model consistency checking. Epsilon HUTN is implemented using the model-to-model transformation,model-to-text transformation and model validation languages of Epsilon. Although any languages for model-to-model transformation (M2M), model-to-text transformation (M2T) and model validation could have been used, Epsilon's existing domain-specific languages are tightly

integrated and inter-operable, making it feasible to chain model management operations together to implement Epsilon HUTN.

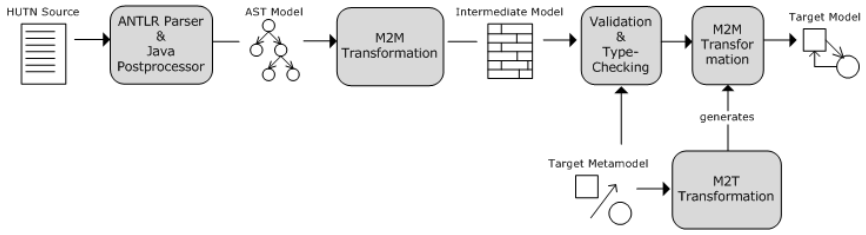


Figure 5.6: The architecture of Epsilon HUTN.

Figure 5.6 outlines the workflow through Epsilon HUTN, from HUTN source text to instantiated target model. The HUTN model specification is parsed to an abstract syntax tree using a HUTN parser specified in ANTLR [Parr 2007]. From this, a Java postprocessor is used to construct an instance of a simple AST metamodel (which comprises two meta-classes, Tree and Node). Using ETL, M2M transformations are then applied to produce an instance of the generic metamodel discussed in Section 5.1. Finally, a M2T transformation on the target metamodel, specified in EGL, produces a further M2M transformation, from the generic metamodel to the target model.

The workflow uses an extension of the generic metamodel defined in Section 5.1. Because the HUTN specification allows the use of packages, an extra element, `PackageObject`, was added to the generic metamodel. A `PackageObject` has a type, an optional identifier and contains any number of `Objects`. To avoid confusion with `PackageObjects`, the `Object` class in the generic metamodel was renamed to `ClassObject`.

Using two M2M transformation stages with the (extended) generic metamodel as an intermediary has two advantages. Firstly, the form of the AST metamodel is not suited to a one-step transformation. There is a mismatch between the features of the AST metamodel and the needs of the target model – for example, between the Node class in the AST metamodel and classes in the target metamodel. If a one-step transformation were used, each transformation rule would need a lengthly guard statement, which is hard to understand and verify. Secondly, Section 5.1 discussed a mechanism for binding XMI to the generic metamodel, which can be used in conjunction with the latter half of the Epsilon HUTN workflow (Figure 5.6) to generate HUTN from XMI. This process is discussed further in Section 5.2.3.

Throughout the remainder of this section, instances of the generic metamodel producing during the execution of the HUTN workflow are termed an *intermediate model*. The two M2M transformations are now discussed in depth, along with a model validation phase which is performed prior to the second transformation.

AST Model to Intermediate Model Epsilon HUTN uses ETL for specifying M2M transformation. One of the transformation rules from Epsilon HUTN is shown in Listing 5.9. The rule transforms a name node in the AST model (which could represent a package or a class object) to a package object in the intermediate model. The guard (line 5) specifies that a name node will only be transformed to a package object if the node has no parent (i.e. it is a top-level node, and hence a package rather than a class). The body of the rule states that the type, line number and column number of the package are determined from the text, line and column attributes of the node object. On line 11, a containment slot is instantiated to hold the children of this package object. The children of the node object are transformed to the intermediate model (using a built-in method, `equivalent()`), and added to the containment slot.

```
1  rule NameNode2PackageObject
2     transform n : AntlrAst!NameNode
3     to p : Intermediate!PackageObject {
4
5     guard : n.parent.isUndefined()
6
7     p.type := n.text;
8     p.line := n.line;
9     p.col := n.column;
10
11    var slot := new Intermediate!ContainmentSlot;
12    for (child in n.children) {
13       slot.objects.add(child.equivalent());
14    }
15    if (slot.objects.notEmpty()) {
16       p.slots.add(slot);
17    }
18 }
```

Listing 5.9: Transformation rule (in ETL) to convert AST nodes to package objects.

Intermediate Model Validation An advantage of the two-stage transformation is that contextual analysis can be specified in an abstract manner – that is, without having to express the traversal of the AST. This gives clarity and minimises the amount of code required to define syntatic constraints.

```
1  context ClassObject {
2     constraint IdentifiersMustBeUnique {
3        guard: self.id.isDefined()
4        check: ClassObject.allInstances()
5                .select(c|c.id = self.id).size() = 1;
```

```
6        message: 'Duplicate identifier: ' + self.id
7    }
8  }
```

Listing 5.10: A constraint (in EVL) to check that all identifiers are unique.

Epsilon HUTN uses EVL [Kolovos *et al.* 2008b] to specify verification, resulting in highly expressive syntactic constraints. An EVL constraint comprises a guard, the logic that specifies the constraint, and a message to be displayed if the constraint is not met. For example, Listing 5.10 specifies the constraint that every HUTN class object has a unique identifier.

In addition to the syntactic constraints defined in the HUTN specification, the conformance constraints described in Section 5.1 are executed on the model at this stage. For this purpose, the conformance constraints are specified in EVL.

Intermediate Model to Target Model Because the contextual analysis is performed on the intermediate model, models conform to the target metamodel. In generating the target model from the intermediate model (Figure 5.6), the transformation uses information from the target metamodel, such as the names of classes and features. A typical approach to this category of problem is to use a higher-order transformation on the target metamodel to generate the desired transformation. Epsilon HUTN uses a different approach: the transformation to the target model is produced by executing an EGL template on the target metamodel. EGL is a template-based text generation language. `[% %]` tag pairs are used to denote dynamic sections, which may produce text when executed. Any code not enclosed in a `[% %]` tag pair is included verbatim in the generated text.

Listing 5.11 is the EGL template for a M2T transformation on the target metamodel; it generates the M2M transformation used for generating the target model. The loop beginning on line 1 iterates over each meta-class in the metamodel, producing a transformation rule to generate target model instances of that meta-class from class objects in the intermediate model. The template guard (line 6) specifies that only class objects of the same type as the meta-class be transformed by the current rule. For the body of the rule the template iterates over each structural feature of the current meta-class, and generates appropriate transformation code for populating the values of each structural feature from the slots on the class object in the intermediate model. The template body is omitted in Listing 5.11 because it contains a large amount of code for interacting with EMF, which is not relevant to this discussion.

```
1  [% for (class in EClass.allInstances()) { %]
2  rule Object2[%=class.name%]
3    transform o : Intermediate!ClassObject
4    to t : Model![%=class.name%] {
```

```
5
6     guard: o.type = '[%=class.name%]'
7
8     -- body omitted
9   }
10  [% } %]
```

Listing 5.11: Initial sections of the template (in EGL) for generating rules (in ETL) to instantiate classes of the target metamodel.

Presently, Epsilon HUTN can be used only to generate EMF models. Support for other modelling languages, such as MDR, would require different transformations between intermediate and target model. In other words, for each target modelling language, a new EGL template would be required. The transformation from AST to intermediate model is independent of the target modelling language and would not need to change.

5.2.3 Migration with HUTN

Epsilon HUTN uses the generic metamodel (from Section 5.1) as an intermediary, facilitating transformation from XMI to HUTN (i.e. the inverse of the transformation discussed above): XMI is parsed to produce an instance of the generic metamodel, and an unparser (implemented using the visitor design pattern [Gamma *et al.* 1995]) generates HUTN source. In this manner, HUTN can be generated for any XMI document, regardless of whether the model described by the XMI conforms to its metamodel.[2]

To demonstrate the way in which HUTN can be used to perform migration, the exemplar XMI shown in Listing 5.1 is represented using HUTN in Listing 5.12. Recall that the XMI describes three Persons, Franz, Julie and Hermann. Julie and Hermann are the mother and father of Franz.

```
1  Persons "kafkas" {
2     Person "Franz" { name: "Franz" }
3     Person "Julie" { name: "Julie" }
4     Person "Hermann" { name: "Hermann" }
5
6     Person "Franz" mother Person "Julie";
7     Person "Franz" father Person "Hermann";
8  }
```

Listing 5.12: HUTN for people with mothers and fathers.

Note that, by using a configuration file to specify that a Person's name is taken from its identifier, the body of the Person objects could be omitted.

[2]TODO: Somewhere, I need to discuss loss of information. (e.g. model element type information when a metaclass is removed)

If the Persons metamodel now evolves such that mother and father are merged to form a parents reference, Epsilon HUTN reports conformance problems on the HUTN document, as illustrated by the screenshot in Figure 5.7.

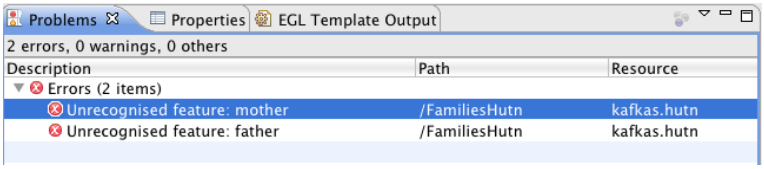


Figure 5.7: Conformance problem reporting in Epsilon HUTN.

Resolving the conformance problems requires the user to change the feature named in the infix associations from mother (father) to parents. The Epsilon HUTN development tools provide content assistance, which might be useful in this situation. Listing 5.13 shows a HUTN document that conforms to the metamodel defining parents rather than mother and father.

```
1  Persons "kafkas" {
2      Person "Franz" { name: "Franz" }
3      Person "Julie" { name: "Julie" }
4      Person "Hermann" { name: "Hermann" }
5
6      Person "Franz" parents Person "Julie";
7      Person "Franz" parents Person "Hermann";
8  }
```

Listing 5.13: HUTN for people with parents.

5.2.4 Limitations

Notwithstanding the power of genericity, there are situations where a metamodel-specific concrete syntax is preferable. An example of where HUTN is unhelpful arose when developing a metamodel for the recording of failure behaviour of components in complex systems, based on the work of [Wallace 2005].

Failure behaviours comprise a number of expressions that specify how each component reacts to system faults, and there is an established concrete syntax for expressing failure behaviours. The failure syntax allows various shortcuts, such as the use of underscore to denote a wildcard. For example, the syntax for a possible failure behaviour of a component that receives input from two other components (on the left-hand side of the expression), and produces output for a single component is denoted:

$$(\{\_\}, \{\_\}) \rightarrow (\{late\}) \tag{5.1}$$

The above expression is written using a domain-specific syntax. In HUTN, the specification of these behaviours is less concise. For example, Listing 5.14 gives the HUTN syntax for failure behaviour (5.1), above.

```
1  Behaviour {
2     lhs: Tuple {
3        contents: IdentifierSet { contents: Wildcard {} },
4                  IdentifierSet { contents: Wildcard {} }
5     }
6
7     rhs: Tuple {
8        contents: IdentifierSet { contents: Fault "late" {} }
9     }
10 }
```

Listing 5.14: Failure behaviour specified in HUTN.

The domain-specific syntax exploits two characteristics of failure expressions to achieve a compact notation. Firstly, structural domain concepts are mapped to symbols: tuples to parentheses and identifier sets to braces. Secondly, little syntactic sugar is needed for many domain concepts, as they define only one feature: a fault is referred to only by its name, the contents of identifier sets and tuples are separated using only commas.

In general, HUTN is less concise than a domain-specific syntax for metamodels containing a large number of classes with few attributes, and in cases where most attributes are used to define structural relationships among concepts. However, there might still be benefits from using HUTN in such cases, if the metamodel is likely to be modified frequently, of it the model does not yet have a formal metamodel.

5.2.5 Summary

In this section, HUTN was introduced and its syntax described. An implementation of HUTN for EMF, built atop Epsilon, was discussed. Integration of HUTN for the metamodel-independent syntax discussed in Section 5.1 facilitates user-driven co-evolution with a textual modelling notation other than XMI, as demonstrated by the example above. The remainder of this chapter focuses on developer-driven co-evolution, in which model migration strategies are executable.

5.3 Analysis of Languages used for Migration

Section 4.2.3 discussed existing approaches to model migration, highlighting variation in the languages used for specifying migration strategies. In this section, migration strategy languages are compared, using the example

of metamodel evolution given in Section 5.3.1. From this comparison, requirements for a domain-specific language for specifying and executing model migration strategies are derived (Section 5.3.3, and an implementation is described in the sequel. The work described in this section has been published in [Rose *et al.* 2010e].

5.3.1 Co-Evolution Example

Throughout this section, the following example of an evolution of a Petri net metamodel is used to discuss co-evolution and model migration. The same example has been used previously in co-evolution literature [Cicchetti *et al.* 2008, Garcés *et al.* 2009, Wachsmuth 2007].

In Figure 5.8(a), a Petri `Net` comprises `Places` and `Transitions`. A `Place` has any number of `src` or `dst` `Transitions`. Similarly, a `Transition` has at least one `src` and `dst` `Place`. In this example, the metamodel in Figure 5.8(a) is to be evolved so as to support weighted connections between `Places` and `Transitions` and between `Transitions` and `Places`.

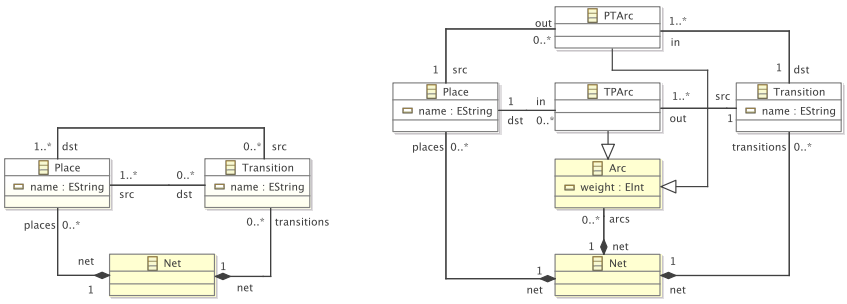
The evolved metamodel is shown in Figure 5.8(b). `Places` are connected to `Transitions` via instances of `PTArc`. Likewise, `Transitions` are connected to `Places` via `TPArc`. Both `PTArc` and `TPArc` inherit from `Arc`, and therefore can be used to specify a `weight`.

Models that conformed to the original metamodel might not conform to the evolved metamodel. The following strategy can be used to migrate models from the original to the evolved metamodel:

1. For every instance, t, of `Transition`:

   For every `Place`, s, referenced by the `src` feature of t:

   Create a new instance, arc, of `PTArc`.



(a) Original metamodel. (b) Evolved metamodel.

Figure 5.8: Exemplar metamodel evolution. (Shading is irrelevant). Taken from [Rose *et al.* 2010e].

Set s as the `src` of arc.

Set t as the `dst` of arc.

Add arc to the `arcs` reference of the `Net` referenced by t.

For every `Place`, d, referenced by the `dst` feature of t:

Create a new instance, arc, of `TPArc`.

Set t as the `src` of arc.

Set d as the `dst` of arc.

Add arc to the `arcs` reference of the `Net` referenced by t.

2. And nothing else changes.

Using the above example, the existing approaches for specifying and executing model migration strategies are now compared.

5.3.2 Existing Approaches

Using the above example, the existing approaches for specifying and executing model migration strategies are now compared. From this comparison, the strengths and weakness of each approach are highlighted and requirements for a model migration language are synthesised in the sequel.

Manual Specification with Model-to-Model Transformation

A model-to-model transformation specified between original and evolved metamodel can be used for performing model migration. Part of the model migration for the Petri nets metamodel is codified with the Atlas Transformation Language (ATL) [Jouault & Kurtev 2005] in Listing 5.15. Rules for migrating `Places` and `TPArcs` have been omitted for brevity, but are similar to the `Nets` and `PTArcs` rules.

In ATL, *rule*s transform source model elements (specified using the `from` keyword) to target model elements (specified using `to` keyword). For example, the `Nets` rule on line 1 of Listing 5.15 transforms an instance of `Net` from the original (source) model to an instance of `Net` in the evolved (target) model. The source model element (the variable `o` in the `Net` rule) is used to populate the target model element (the variable `m`). ATL allows rules to be specified as *lazy* (not scheduled automatically and applied only when called by other rules).

In model transformation, [Czarnecki & Helsen 2006] identifies two common categories of relationship between source and target model, *new target* and *existing target*. In the former, the target model is constructed afresh by the execution of the transformation, while in the latter, the target model contains the same data as the source model before the transformation is executed. ATL supports both new and existing target relationships (the latter is termed

a refinement transformation). However, ATL refinement transformations may only be used when the source and target metamodel are the same, as is typical for existing target transformations.

```
1  rule Nets {
2    from o : Before!Net
3    to m : After!Net ( places <- o.places, transitions <- o.transitions )
4  }
5
6  rule Transitions {
7    from o : Before!Transition
8    to m : After!Transition (
9       name <- o.name,
10      "in" <- o.src->collect(p | thisModule.PTArcs(p,o)),
11      out <- o.dst->collect(p | thisModule.TPArcs(o,p))
12    )
13 }
14
15 unique lazy rule PTArcs {
16   from place : Before!Place, destination : Before!Transition
17   to ptarcs : After!PTArc (
18      src <- place, dst <- destination, net <- destination.net
19    )
20 }
```

Listing 5.15: Fragment of the Petri nets model migration in ATL

In model migration, source and target metamodels differ, and hence existing target transformations cannot be used to specify model migration strategies. Consequently, model migration strategies are specified with new target model-to-model transformation languages, and often contain sections for copying from original to migrated model those model elements that have not been affected by metamodel evolution. For the Petri nets example, the `Nets` rule (in Listing 5.15) and the `Places` rule (not shown) exist only for this reason.

The `Transitions` rule in Listing 5.15 codifies in ATL the migration strategy described previously. The rule is executed for each `Transition` in the original model, `o`, and constructs a `PTArc` (`TPArc`) for each reference to a `Place` in `o.src` (`o.dst`). Lazy rules must be used to produce the arcs to prevent circular dependencies with the `Transitions` and `Places` rules. Here, ATL, a typical rule-based transformation language, is considered and model migration would be similar in QVT. With Kermeta, migration would be specified in an imperative style using statements for copying `Nets`, `Places` and `Transitions`, and for creating `PTArcs` and `TPArcs`.

Manual Specification with Ecore2Ecore Mapping

Hussey and Paternostro [Hussey & Paternostro 2006] explain the way in which integration with the model loading mechanisms of the Eclipse Modeling Framework (EMF) [Steinberg *et al.* 2008] can be used to perform model migration. In this approach, the default metamodel loading strategy is augmented with model migration code.

Because EMF binds models to their metamodel (discussed in Section 4.2.1), EMF cannot use an evolved metamodel to load an instance of the original metamodel. Therefore, Hussey and Paternostro's approach requires the metamodel developer to provide a mapping between the metamodelling language of EMF, Ecore, and the concrete syntax used to persist models, XMI. Mappings are specified using a tool that can suggest relationships between source and target metamodel elements by comparing names and types.

Model migration is specified on the XMI representation of the model and hence presumes some knowledge of the XMI standard. For example, in XMI, references to other model elements are serialised as a space delimited collection of URI fragments [Steinberg *et al.* 2008]. Listing 5.16 shows a section of the Ecore2Ecore model migration for the Petri net example presented above. The method shown converts a `String` containing URI fragments to a `Collection` of `Places`. The method is used to access the `src` and `dst` features of `Transition`, which no longer exist in the evolved metamodel and hence are not loaded automatically by EMF. To specify the migration strategy for the Petri nets example, the metamodel developer must know the way in which the `src` and `dst` features are represented in XMI. The complete listing, not shown here, exceeds 200 lines of code.

```
1  private Collection<Place> toCollectionOfPlaces
2  (String value, Resource resource) {
3
4    final String[] uriFragments = value.split("␣");
5    final Collection<Place> places = new LinkedList<Place>();
6
7    for (String uriFragment : uriFragments) {
8      final EObject eObject = resource.getEObject(uriFragment);
9      final EClass place = PetriNetsPackage.eINSTANCE.getPlace();
10
11     if (eObject == null || !place.isInstance(eObject))
12       // throw an exception
13
14     places.add((Place)eObject);
15   }
16
17   return places;
```

```
18  }
```

Listing 5.16: Java method for deserialising a reference.

Operator-based Co-evolution with COPE

Operator-based approaches to managing co-evolution, such as COPE [Herrmannsdoerfer *et al.* 200 provide a library of *co-evolutionary operators*. Each co-evolutionary operator specifies both a metamodel evolution and a corresponding model migration strategy. For example, the "Make Reference Containment" operator from COPE [Herrmannsdoerfer *et al.* 2009b] evolves the metamodel such that a non-containment reference becomes a containment reference and migrates models such that the values of the evolved reference are replaced by copies. By composing co-evolutionary operators, metamodel evolution can be performed and a migration strategy can be generated without writing any code.

To perform metamodel evolution using an operator-based approach, the library of co-evolutionary operators must be integrated with tools for editing metamodels. COPE provides integration with the EMF tree-based metamodel editor. Operators may be applied to an EMF metamodel, and a record of changes tracks their application. Once metamodel evolution is complete, a migration strategy can be generated automatically from the record of changes. The migration strategy is distributed along with the updated metamodel, and metamodel users choose when to execute the migration strategy on their models.

To be effective, operator-based approaches must provide a rich yet navigable library of co-evolutionary operators, as discussed in Section 4.2.3. To this end, COPE allows model migration strategies to be specified manually when no co-evolutionary operator is appropriate. Rather than use either of the two manual specification approaches discussed above (model-to-model transformation and Ecore2Ecore mapping), COPE employs a fundamentally different approach using an existing target transformation.

As discussed above, existing target transformations cannot be used for specifying model migration strategies as the source (original) and target (evolved) metamodels differ. However, models can be structured independently of their metamodel using a *metamodel-independent representation*. Figure 5.9 shows a simplification of the metamodel-independent representation used by COPE. By using a metamodel-independent representation of models as an intermediary, an existing target transformation can be used for performing model migration when the migration strategy is specified in terms of the metamodel-independent representation. Further details of this technique are given in [Herrmannsdoerfer *et al.* 2009b].

Listing 5.17 shows the COPE model migration strategy for the Petri net example given above[3]. Most notably, slots for features that no longer exist

[3]In Listing 5.17, some of the concrete syntax has been changed in the interest of read-

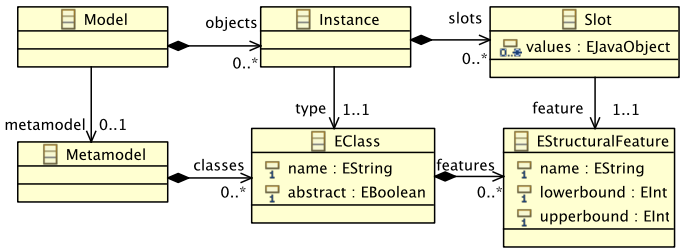


Figure 5.9: Simplification of the metamodel-independent representation used by COPE, based on [Herrmannsdoerfer *et al.* 2009b].

must be explicitly `unset`. In Listing 5.17, slots are `unset` on four occasions, once for each feature that exists in the original metamodel but not the evolved metamodel. Namely, these features are: `src` and `dst` of `Transition` and of `Place`. Failing to `unset` slots that do not conform with the evolved metamodel causes migration to fail with an error.

```
1  for (transition in Transition.allInstances) {
2    for (source in transition.unset('src')) {
3      def arc = petrinets.PTArc.newInstance()
4      arc.src = source; arc.dst = transition;
5      arc.net = transition.net
6    }
7
8    for (destination in transition.unset('dst')) {
9      def arc = petrinets.TPArc.newInstance()
10     arc.src = transition; arc.dst = destination;
11     arc.net = transition.net
12   }
13 }
14
15 for (place in Place.allInstances) {
16   place.unset('src'); place.unset('dst');
17 }
```

Listing 5.17: Petri nets model migration in COPE

5.3.3 Requirements Identification

By analysing existing approaches to managing developer-driven co-evolution, requirements were derived for a domain-specific language for specifying and executing model migration. The derivation of the requirements is now summarised, by considering two dimensions: the source-target relationship of the

ability.

language used for specifying migration strategies and the way in which models are represented during migration.

Source-Target Relationship

New target transformation languages (Section 5.3.2) require code for explicitly copying from the original to the evolved metamodel those model elements that are unaffected by the metamodel evolution. In contrast, model migration strategies written in COPE (Section 5.3.2) must explicitly unset any data that is not to be copied from the original to the migrated model. The Ecore2Ecore approach (Section 5.3.2) does not require explicit copying or unsetting code. Instead, the relationship between original and evolved metamodel elements is captured in a mapping model specified by the metamodel developer. The mapping model can be configured by hand or, in some cases, automatically derived.

In each case, extra effort is required when defining a migration strategy due to the way in which the co-evolution approach relates source (original) and target (migrated) model elements. This observation led to the following requirement: *The migration language must* ***automatically*** *copy every model element that conforms to the evolved metamodel from original to migrated model, and must not automatically copy any model element that does not conform to the evolved metamodel from original to migrated model.*

Model Representation

When using the Ecore2Ecore approach, model elements that do not conform to the evolved metamodel are accessed via XMI. Consequently, the metamodel developer must be familiar with XMI and must perform tasks such as dereferencing URI fragments (Listing 5.16) and type conversion. With COPE and the Epsilon Transformation Language, models are loaded using a modelling framework (and so migration strategies need not be concerned with the representation used to store models). Consequently, the following requirement was identified: *The migration language must not expose the underlying representation of original or migrated models.*

To apply co-evolution operators, COPE requires the metamodel developer to use a specialised metamodel editor, which can manipulate only metamodels defined with EMF. Like, the Ecore2Ecore approach, COPE can be used only to manage co-evolution for models and metamodels specified with EMF. Tight coupling to EMF allows the Ecore2Ecore approach to schedule migration automatically, during model loading. To better support integration with modelling frameworks other than EMF, the following requirement was derived: *The migration language must be loosely coupled with modelling frameworks and must not assume that models and metamodels will be represented in EMF.*

5.4 Epsilon Flock: A Model Migration Language

Driven by the analysis presented above, a domain-specific language for model migration, Epsilon Flock (subsequently referred to as Flock), was designed and implemented. Section 5.4.1 discusses the principle tenets of Flock, including the way in which automatically maps each element of the original model to an equivalent element of the migrated model using a novel conservative copying algorithm and user-defined migration rules. In Section 5.4.2, Flock is demonstrated via application to two examples of model migration. The work described in this section has been published in [Rose *et al.* 2010e].

5.4.1 Design and Implementation

Flock is a rule-based transformation language that mixes declarative and imperative parts. Its style is inspired by hybrid model-to-model transformation languages such as the Atlas Transformation Language [Jouault & Kurtev 2005] and the Epsilon Transformation Language [Kolovos *et al.* 2008a]. Flock has a compact syntax. Much of its design and implementation is focused on the runtime. The way in which Flock relates source to target elements is novel; it is neither a new nor an existing target relationship. Instead, elements are copied conservatively, as described in Section 5.4.1.

Like Epsilon HUTN (5.2.2), Flock is built atop Epsilon, which was described in Section 2.3.2. In particular, Flock uses the model connectivity layer and the object language of Epsilon. The former is used to decouple migration from the representation of models and to provide compatibility with several modelling frameworks, while the latter provides a language for specifying user-defined migration rules, which are now discussed.

Abstract Syntax

As illustrated by Figure 5.10, Flock migration strategies are organised into modules (`FlockModule`), which inherit from EOL modules (`EolModule`), which provides support for module reuse with import statements and user-defined operations. Modules comprise any number of rules (`Rule`). Each rule has an original metamodel type (`originalType`) and can optionally specify a `guard`, which is either an EOL statement or a block of EOL statements. `MigrateRules` must specify an evolved metamodel type (`evolvedType`) and/or a `body` comprising a block of EOL statements.

Concrete Syntax

Listing 5.18 shows the concrete syntax of migrate and delete rules. All rules begin with a keyword indicating their type (either `migrate` or `delete`), followed by the original metamodel type. Guards are specified using the `when`

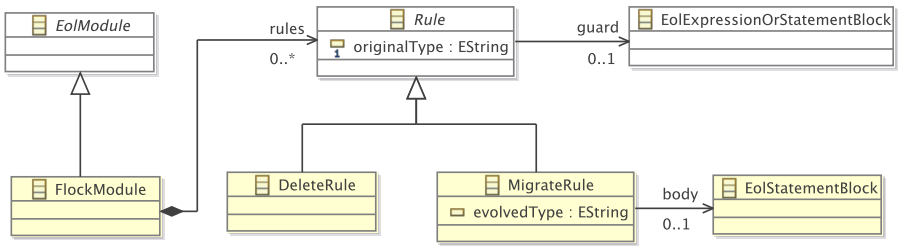


Figure 5.10: The abstract syntax of Flock.

```
1  migrate <originalType> (to <evolvedType>)?
2  (when (:<eolExpression>)|({<eolStatement>+}))? {
3    <eolStatement>*
4  }
5
6  delete <originalType>
7  (when (:<eolExpression>)|({<eolStatement>+}))?
```

Listing 5.18: Concrete syntax of migrate and delete rules.

keywords. Migrate rules may also specify an evolved metamodel type using the `to` keyword and a `body` as a (possibly empty) sequence of EOL statements.

Note there is presently no create rule. In Flock, the creation of new model elements is usually encoded in the imperative part of a migrate rule specified on the containing type.

Execution Semantics

A Flock module has the following behaviour when executed:

1. For each original model element, `e`:

   Identify an applicable rule, `r`. To be applicable for `e`, a rule must have as its original type the metaclass (or a supertype of the metaclass) of `e` and the guard part of the rule must be satisfied by `e`.

   When no rule can be applied, a default rule is used, which has the metaclass of `e` as its original type, and an empty body.

2. For each mapping between original model element, `e`, and applicable delete rule, `r`:

   Do nothing.

3. For each mapping between original model element, `e`, and applicable migrate rule, `r`:

Create an equivalent model element, `e'` in the migrated model. The metaclass of `e'` is determined from the `evolvedType` (or the `originalType` when no `evolvedType` has been specified) of `r`.

Copy the data contained in `e` to `e'` (using the *conservative copy* algorithm described in the sequel).

4. For each mapping between original model element, `e`, applicable migrate rule, `r`, and equivalent model element, `e'`:

   Execute the body of `r` binding `e` and `e'` to variables named `original` and `migrated`, respectively.

Conservative Copying

Flock contributes an algorithm, termed *conservative copy*, that copies model elements from original to migrated model only when those model elements conform to the evolved metamodel. Because of its conservative copy algorithm, Flock is a hybrid of new target and existing target transformation languages. This section discusses the conservative copying algorithm in more detail.

The algorithm operates on an original model element, `o`, and its equivalent model element in the migrated model, `e`. When `o` has no equivalent in the migrated model (for example, when a metaclass has been removed and the migration strategy specifies no alternative metaclass), `o` is not copied to the migrated model. Otherwise, conservative copy is invoked for `o` and `e`, proceeding as follows:

- For each metafeature, `f` for which `o` has specified a value

  Locate a metafeature in the evolved metamodel with the same name as `f` for which `e` may specify a value.

  When no equivalent metafeature can be found, do nothing.

  Otherwise, copy to the migrated model the original value (`o.f`) only when it conforms to the equivalent metafeature

The definition of conformance varies over modelling frameworks. Typically, conformance between a value, `v`, and a feature, `f`, specifies at least the following constraints:

- The size of `v` must be greater than or equal to the lowerbound of `f`.
- The size of `v` must be less than or equal to the upperbound of `f`.
- The type of `v` must be the same as or a subtype of the type of `f`.

Epsilon includes a model connectivity layer (EMC), which provides a common interface for accessing and persisting models. Currently, EMC provides

drivers for several modelling frameworks, permitting management of models defined with EMF, the Metadata Repository (MDR), Z or XML. To support migration between metamodels defined in heterogenous modelling frameworks, EMC was extended during the development of Flock. The connectivity layer now provides a conformance checking service. Each EMC driver was extended to include conformance checking semantics specific to its modelling framework. Flock implements conservative copy by delegate conformance checking responsibilities to EMC.

Finally, some categories of model value must be converted before being copied from the original to the migrated model. Again, the need for and semantics of this conversion varies over modelling frameworks. Reference values typically require conversion before copying. In this case, the mappings between original and migrated model elements maintained by the Flock runtime can be used to perform the conversion. In other cases, the target modelling framework must be used to perform the conversion, such as when EMF enumeration literals are to be copied.

Development and User Tools

As discussed in Section 4.2, models and metamodels are typically kept separate. Flock migration strategies can be distributed by the metamodel developer in two ways. An extension point defined by Flock provides a generic user interface for migration strategy execution. Alternatively, metamodel developers can elect to build their own interface, delegating execution responsibility to `FlockModule`. We anticipate the latter to be useful for production environments using model or source code management repositories.

5.4.2 Examples

Flock is now demonstrated using two examples of model migration. Listing 5.19 illustrates the Flock migration strategy for the Petri net example introduced above and is included for direct comparison with other approaches. An additional, larger example is presented based on changes made to UML class diagrams between versions 1.5 and 2.0 of the UML specification.

Petri Nets in Flock

The exemplar Petri net metamodel evolution is now revisited to demonstrate the basic functionality of Flock. In Listing 5.19, `Nets` and `Places` are migrated automatically. Unlike the ATL migration strategy (Listing 5.15), no explicit copying rules are required. Compared to the COPE migration strategy (Listing 5.17), the Flock migration strategy does not explicitly unset the original `src` and `dst` features of `Transition`.

```
1 migrate Transition {
```

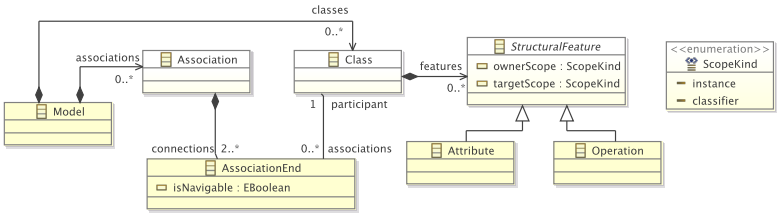
```
2   for (source in original.src) {
3     var arc := new Migrated!PTArc;
4     arc.src := source.equivalent(); arc.dst := migrated;
5     arc.net := original.net.equivalent();
6   }
7
8   for (destination in original.dst) {
9     var arc := new Migrated!TPArc;
10    arc.src := migrated; arc.dst := destination.equivalent();
11    arc.net := original.net.equivalent();
12  }
13 }
```

Listing 5.19: Petri nets model migration in Flock

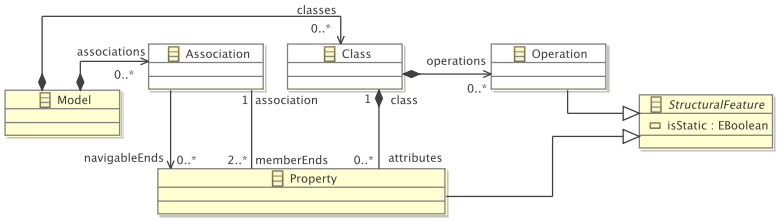
Petri Nets in Flock

Figure 5.11 illustrates a subset of the changes made between UML 1.5 and UML 2.0. Only class diagrams are considered, and features that did not change are omitted. In Figure 5.11(a), association ends and attributes are specified explicitly and separately. In Figure 5.11(b), the `Property` class is used instead. The Flock migration strategy (Listing 5.20) for Figure 5.11 is now discussed.

```
1  migrate Association {
2    migrated.memberEnds := original.connections.equivalent();
3  }
4
5  migrate Class {
6    var fs := original.features.equivalent();
7    migrated.operations := fs.select(f|f.isKindOf(Operation));
8    migrated.attributes := fs.select(f|f.isKindOf(Property));
9    migrated.attributes.addAll(original.associations.equivalent())
10 }
11
12 delete StructuralFeature when: original.targetScope <> #instance
13
14 migrate Attribute to Property {
15   if (original.ownerScope = #classifier) {
16     migrated.isStatic = true;
17   }
18 }
19 migrate Operation {
20   if (original.ownerScope = #classifier) {
21     migrated.isStatic = true;
22   }
23 }
```



(a) Original metamodel.



(b) Evolved metamodel.

Figure 5.11: Exemplar UML metamodel evolution. (Shading is irrelevant).

```
24
25 migrate AssociationEnd to Property {
26   if (original.isNavigable) {
27     original.association.equivalent().navigableEnds.add(migrated)
28   }
29 }
```

Listing 5.20: UML model migration in Flock

Firstly, `Attributes` and `AssociationEnds` are now modelled as `Properties` (lines 16 and 28). In addition, the `Association#navigableEnds` reference replaces the `AssociationEnd#isNavigable` attribute; following migration, each navigable `AssociationEnd` must be referenced via the `navigableEnds` feature of its `Association` (lines 29-31).

In UML 2.0, `StructuralFeature#ownerScope` has been replaced by `#isStatic` (lines 17-19 and 23-25). The UML 2.0 specification states that `ScopeKind#classifier` should be mapped to true, and `#instance` to false.

The UML 1.5 `StructuralFeature#targetScope` feature is no longer supported in UML 2.0, and no migration path is provided. Consequently, line 14 deletes any model element whose `targetScope` is not the default value.

Finally, `Class#features` has been split to form `Class#operations` and `#attributes`. Lines 8 and 10 partition features on the original `Class`. `Class#associations` has been removed in UML 2.0, and `AssociationEnds` are instead stored in `Class#attributes` (line 11).

Comparison

Table 5.1 illustrates several characterising differences between Flock and the related approaches presented in Section 5.3.1. Due to its conservative copying algorithm, Flock is the only approach to provide both automatic copying and unsetting. Automatic copying is significant for metamodel evolutions with a large number of unchanging features.

All of the approaches considered in Table 5.1 support EMF, arguably the most widely used modelling framework. The Ecore2Ecore approach, however, requires migration to be encoded at the level of the underlying model representation XMI. Both Flock and ATL support other modelling technologies, such as MDR and XML. However, ATL does not automatically copy model elements that have not been affected by metamodel changes. Therefore, migration between models of different technologies with ATL requires extra statements in the migration strategy to ensure that the conformance constraints of the target technology are satisfied. Because it delegates conformance checking to an EMC driver, Flock requires no such checks.

A more thorough examination of the similarities and differences between Flock and other migration strategy languages is provided in Chapter 6.

5.5 Chapter Summary

The solutions described in this chapter have been released as part of Epsilon in the Eclipse GMT [Eclipse 2008] project, which is the research incubator of arguably the most widely used MDE modelling framework, EMF. By re-using parts of Epsilon, implementations of the solutions were produced more rapidly than if the tools were developed from scratch. In particular, re-using the Epsilon model connectivity layer facilitated interoperability of Flock with several MDE modelling frameworks, which was exploited to manage a practical case of model migration in Section 6.4.

To be completed.

Table 5.1: Properties of model migration approaches

	Automatic copy	**Automatic unset**	**Modelling technologies**
Ecore2Ecore	✓	✗	XMI
ATL	✗	✓	EMF, MDR, KM3, XML
COPE	✓	✗	EMF
Flock	✓	✓	EMF, MDR, XML, Z

Chapter 6

Evaluation

Chapter 5 outlined requirements for and described the implementation of a textual modelling notation and a model migration language. This chapter describes the way in which both solutions were evaluated against their requirements, and against the thesis requirements identified in Chapter 4. A further solution, the metamodel-independent syntax, was developed in Section 5.1 and is not evaluated explicitly in this chapter, as discussed below.

The textual modelling notation (Section 5.2) was developed to address the challenges faced when performing user-driven co-evolution. Using a real-world example of user-driven co-evolution, Section 6.1 demonstrates existing and novel processes for performing manual migration, and indicates the benefits of using the textual modelling notation in the novel process.

A transformation language tailored for model migration, Epsilon Flock (Section 5.4), was developed to allow the specification and execution of model migration strategies. Flock uses a novel algorithm for relating source to target model elements, termed conservative copy. Conservative copy is evaluated against two existing mechanisms for relating source and target model elements in Section 6.2 using quantitive measurements. In addition, Epsilon Flock has been evaluated by comparison with other migration tools (Section 6.3) and with transformation languages (Section 6.4). The developers of other tools have been involved in the evaluation described in Sections 6.3 and 6.4, and their contributions are highlighted. The work presented in this chapter has been published in [Rose *et al.* 2010a, Rose *et al.* 2010d, Rose *et al.* 2010c].

Evaluation of the Metamodel-Indepenent Syntax The metamodel-independent syntax was developed to address the following thesis requirement, identified in Section 4.3: *This thesis must investigate the extension of existing modelling frameworks to support the loading of non-conformant models and conformance checking of models against other metamodels.*

The metamodel-independent syntax was demonstrated to facilitate loading of non-conformant models and conformance checking in Section 5.1.3. Addi-

tionally, the metamodel-independent syntax was used in the implementation of the textual modelling notation, which is evaluated in Section 6.1. As such, no further evaluation of the metamodel-independent syntax is presented in this chapter. Instead, the remainder of the chapter focuses on the evaluation of the textual modelling notation and the model migration language.

6.1 Evaluating User-Driven Co-Evolution

This section evaluates the suitability of the metamodel independent syntax (Section 5.1) and the textual modelling notation (Section 5.2) for performing user-driven co-evolution, a novel process for managing co-evolution that was identified in Chapter 4. User-driven co-evolution was observed in real-world MDE projects in Chapter 4, but no tool support for user-driven co-evolution is reported in the co-evolution literature. The evaluation presented in this section seeks to demonstrate that dedicated structures for user-driven co-evolution can increase the productivity of model migration in a user-driven co-evolution process.

To explore this claim, several approaches to evaluation could have been used. The dedicated structures for user-driven co-evolution are freely available as part of Epsilon, a member of the Eclipse Modeling Project. The productivity benefits of the structures might have been explored by asking users to describe their experiences with the structures. However, the subjective nature of the feedback might have threatened the validity of this evaluation. Alternatively, evaluation might have been performed with a comprehensive user study that measured the time taken for developers to perform model migration with and without the dedicated structures for user-driven co-evolution. However, locating developers and examples of co-evolution for this study was not possible given the time available to perform the evaluation.

Instead, evaluation was conducted by comparing two approaches to user-driven co-evolution using an example of user-driven co-evolution from a real-world MDE project. The first approach uses only those tools available in EMF; while the second approach uses EMF and, in addition, two of the structures presented in Chapter 5, a metamodel-independent syntax and a textual modelling notation. The remainder of this section first recaps Section 4.2.2, which describes the challenges to productivity faced by developers while performing user-driven co-evolution with the Eclipse Modeling Framework (EMF), arguably the most widely-used MDE development environment at present. Subsequently, the two approaches to user-driven co-evolution are demonstrated. The section concludes by comparing the two approaches and highlighting ways in which two of the structures proposed in Chapter 5 might be used to increase the productivity of model migration in a user-driven co-evolution process.

6.1.1 Challenges for Performing User-Driven Co-Evolution

Chapter 4 highlighted two productivity challenges faced by developers while performing user-driven co-evolution in contemporary MDE environments. Firstly, model storage representations have not been optimised for use by humans, and hence user-driven co-evolution can be error-prone and time consuming. Secondly, the multi-pass parsers used to load models in contemporary MDE environments cause user-driven co-evolution to be an iterative process, because not all conformance errors are reported at once. The identification of these productivity challenges led to the derivation of the following research requirement: *This thesis must demonstrate a user-driven co-evolution process that enables the editing of non-conformant models without directly manipulating the underlying storage representation and provides a conformance report for the original model and evolved metamodel.*

Two of the structures presented in Chapter 5 provide the foundation for fulfilling the above research requirement. The first, a metamodel-independent syntax, facilitates the conformance checking of a model against any metamodel. The second structure, a textual modelling notation called HUTN, allows models to be managed in a format that is reputedly easier for humans to use than XMI, the canonical model storage format [OMG 2004].

To fulfil the above research requirement, this section uses the metamodel-independent syntax and the textual modelling notation to demonstrate that user-driven co-evolution can be performed without encountering the challenges to productivity described above. To this end, an example of co-evolution is used to show the way in which user-driven co-evolution might be achieved with and without the metamodel-independent syntax and the textual modelling notation described in Chapter 5.

6.1.2 Co-Evolution Example

The remainder of this section uses a co-evolution example taken from collaborative work with Adam Sampson, then a Research Associate at the University of Kent. The purpose of the collaboration was to build a prototypical editor for graphical models of programs written in process-oriented programming languages, such as occam-π [Welch & Barnes 2005]. The graphical models would provide a standard notation for describing process-oriented programs.

The graphical model editor was developed using MDE tools and techniques. A metamodel was used to capture the abstract syntax of process-oriented programming languages, and code for a graphical model editor was automatically generated from the metamodel.

The final version of the graphical model editor is shown in Figure 6.1. The editor captures the three primary concepts used to specify process-oriented programs: processes, connection points and channels. Processes, represented as boxes in the graphical notation, are the fundamental building blocks of

a process-oriented program. Channels, represented as lines in the graphical notation, are the mechanism by which processes communicate, and are unidirectional. Connection points, represented as circles in the graphical notation, define the channels on which a process can communicate. Connection points are used to specify the way in which a process can communicate, and can optionally be bound to a channel. Because channels are unidirectional, connection points are either reading (consume messages from the channel) or writing (generate messages on the channel). Reading (writing) connection points are represented as white (black) circles in the graphical notation.

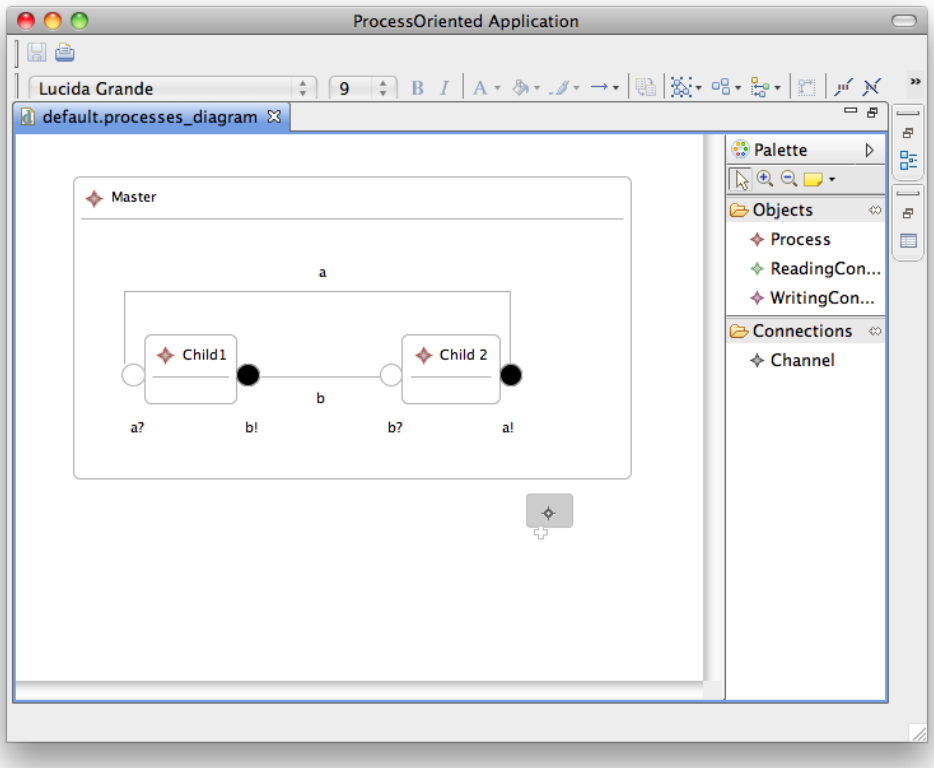


Figure 6.1: Final version of the prototypical graphical model editor.

The graphical model editor was implemented using EMF. The metamodel was specified in Ecore, the metamodelling language of EMF, and the editor was generated from the metamodel using the Graphical Modeling Framework (GMF), an extension to EMF for graphical modelling. Section 2.3 describes in more detail the way in which EMF and GMF can be used to specify metamodels and to generate graphical model editors.

The process-oriented metamodel was developed iteratively, and the six iterations are described in Appendix B. During each iteration, the metamodel

was changed. The remainder of this section uses an example of metamodel changes from the fifth iteration of the project. The way in which development proceeded during that iteration is described in Section B.5 and summarised below.

Feature identification

The purpose of the iteration was to refine the way in which connection points were represented. At the start of the iteration, the graphical model editor could be used to draw processes, channels and connection points. However, no distinction was made between reading and writing connection points.

Figure 6.2 shows an exemplar model represented in the graphical model editor before the iteration began. The model contains two processes (depicted as boxes), `P1` and `P2`, one channel (depicted as a line), `a`, and two connection points (depicted as circles), `a!` and `a?`.

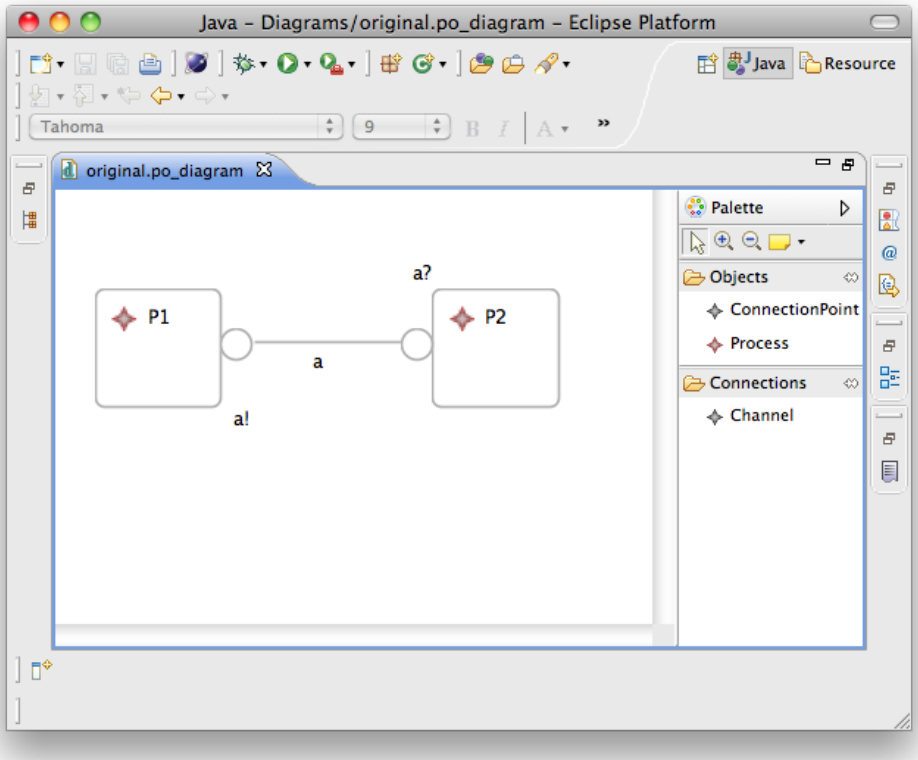


Figure 6.2: The graphical editor at the start of the iteration.

The aim of the iteration was to distinguish between *reading* and *writing* connection points in the graphical notation. The former are used to receive

messages, and the latter to send messages. In Figure 6.2, `a?` is intended to represent a reading connection point, and `a!` a writing connection point. Sampson and the thesis author decided that the editor should be changed so that black circles would be used to represent writing connection points, and white circles to represent reading connection points. At the end of the iteration the model shown in Figure 6.2 would be represented as shown in Figure 6.3. Furthermore, the editor would ensure that `a?` was used only as the reader of a channel, and `a!` only as the writer of a channel. Before the iteration started, the editor did not enforce this constraint.

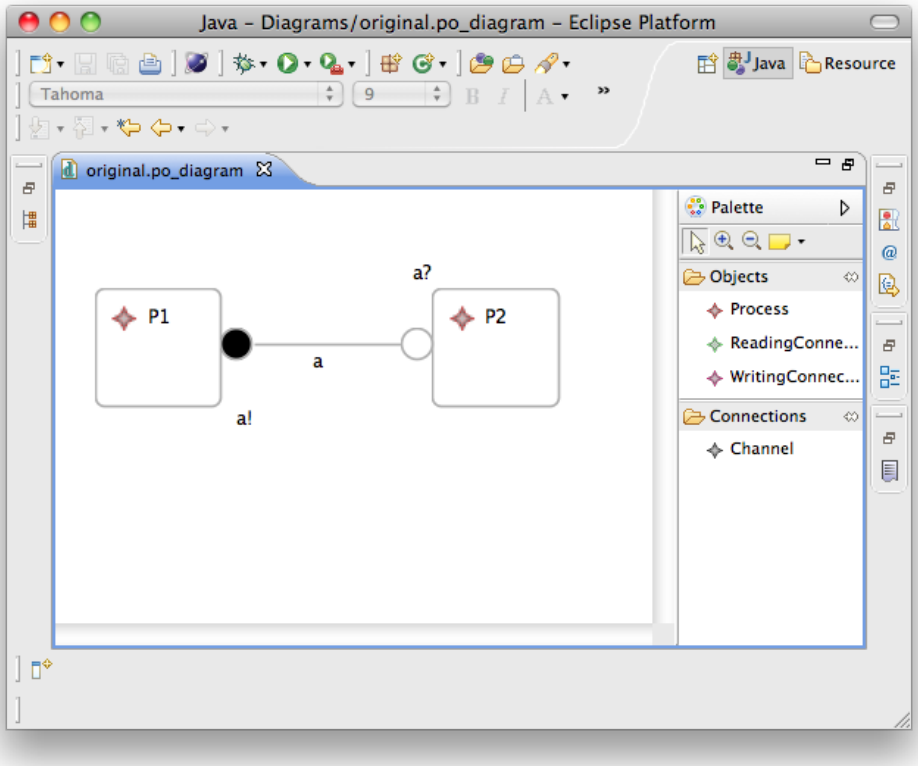


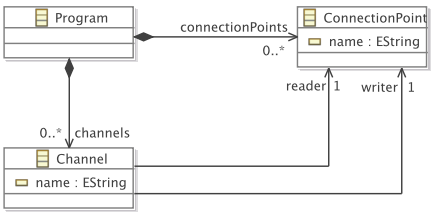
Figure 6.3: The graphical editor at the end of the iteration.

Implementation

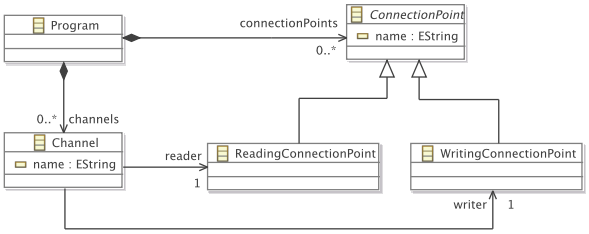
Before the iteration began, the metamodel did not distinguish between reading and writing `ConnectionPoints`. Figure 6.4(a) shows the way in which connection points were modelled at the start of the iteration. When a `ConnectionPoint` was associated with a `Channel`, the `ConnectionPoint`

is specified as a `reader` or a `writer` for that `Channel`, and otherwise the type of a `ConnectionPoint` is not specified.

The way in which connection points were modelled was changed, resulting in the metaclasses shown in Figure 6.4(b). `ConnectionPoint` was made abstract, and two subtypes, `ReadingConnectionPoint` and `WritingConnectionPoint`, were introduced. The `reader` and `writer` references of `Channel` were changed to refer to the new subtypes.



(a) Part of the original metamodel.



(b) Part of the evolved metamodel.

Figure 6.4: Process-oriented metamodel evolution.

Following the metamodel changes, a new version of the graphical editor was generated automatically from the metamodel using GMF. An annotation – not shown in Figure 6.4(b) – on the `WritingConnectionPoints` was used to indicate to GMF that black circles were to be used to represent writing connection points in the graphical notation.

Testing

Testing the new version of the graphical editor highlighted the need for model migration. Attempting to load existing models, such as the one shown in Figure 6.2, caused an error because `ConnectionPoint` was now an abstract class. Any model specifying at least one connection point no longer conformed to the metamodel. Model migration was performed to re-establish conformance and to allow the models to be loaded.

Several models, presented in Appendix B had been constructed when testing previous versions of the graphical editor. The models were used during

each iteration to ensure that any changes had not introduced regressions. Following the metamodel changes described above, the models could no longer be loaded and required migration. A user-driven rather than a developer-driven co-evolution approach was preferred throughout the development of process-oriented editor because only a few small models required migration in each iteration.

The remainder of this section describes the way in which model migration was performed for the changes to the process-oriented metamodel outlined above. The sequel describes the way in which migration was performed during the development of the process-oriented metamodel, without dedicated structures for performing user-driven co-evolution. Section 6.1.4 describes the way in which migration could have been performed using two of the structures presented in Chapter 5. The section concludes by comparing the two approaches.

6.1.3 User-Driven Co-Evolution with EMF

During the development of the process-oriented metamodel, no structures for performing user-driven co-evolution were available. Instead, migration was performed using only those tools available in EMF, as described below.

Migration with EMF involved identifying and fixing conformance errors, using the approach shown in Figure 6.5. When a model is loaded by the graphical editor, EMF automatically reports any conformance problems. Because EMF cannot load non-conformant models, the underlying storage representation of the model, XMI, is edited by hand to reconcile conformance problems. After saving the reconciled XMI to disk, the model is loaded again in the graphical editor and, because EMF uses a multi-pass XMI parser, additional conformance problems might be reported. If further conformance problems are reported, additional changes are made to the XMI. Otherwise, migration is complete.

EMF reports conformance problems when a model is loaded. For the process-oriented metamodel, loading a model involved opening the model with the graphical model editor. For the model shown in Figure 6.2, the conformance problems shown in the bottom pane (and by the error markers in the left-hand margin of the top pane) of Figure 6.6 were reported by EMF. For example, the first conformance problem reported is shown in the tooltip in Figure 6.2, and states that a `ClassNotFoundException` was encountered because the "Class 'ConnectionPoint' is not found or is abstract."

The conformance problems were fixed by editing the XMI shown in Figure 6.6, changing the type of the connection point objects to either a reading or a writing connection point. A reading (writing) connection point was used when the connection point was referenced via the `reader` (`writer`) reference of `Channel`. The reconciled XMI is shown in Figure 6.7. On lines 4 and 7, the connection point model elements have been changed to include `xmi:type`

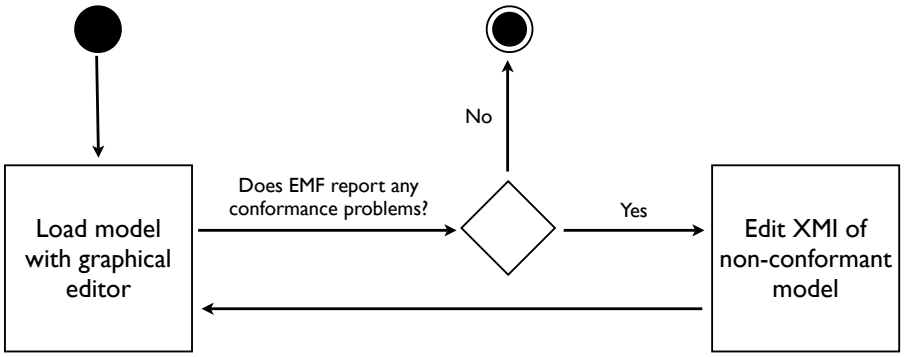


Figure 6.5: User-driven co-evolution with EMF

attributes, which specify whether the connection point should instantiate `ReadingConnectionPoint` or `WritingConnectionPoint`.

Reconciling the conformance problems by editing the XMI required considerable knowledge of the XMI specification. For example, the `xmi:type` attribute is used to specify the type of the connection point model elements. In fact, it must be included for those model elements. However, for the other model elements in Figure 6.7 the `xmi:type` attribute is not necessary, and is omitted. When and how to use the `xmi:type` attribute is discussed further in the sidebar, and is synthesised from the XMI specification [OMG 2007c] and [Steinberg *et al.* 2008]. EMF abstracts away from XMI, and typically users do not interact directly with XMI. Therefore, it may be reasonable to assume that EMF users might not be familiar with XMI, and implementation details such as the `xmi:type` attribute.

The `xmi:type` attribute

In XMI, each model element specifies a value for the `xmi:type` attribute to indicate the metaclass that the model element instantiates. For example, the model element definition on line 4 of Figure 6.7 instantiates the metaclass named `WritingConnectionPoint`. To reduce the size of models on disk, the XMI specification allows type information to be omitted when it can be inferred. For example, line 9 of Figure 6.7 defines a model element that is contained in the `channels` reference of a `Process`. Because the `channels` reference can contain only one type of model element (`Channel`), the `xmi:type` attribute can be omitted, and the type information is inferred from the metamodel.

During the development of the process-oriented editor, some mistakes were made when migrating Sampson's models. For example, the wrong subtype of `ConnectionPoint` was used as the type of several connection point

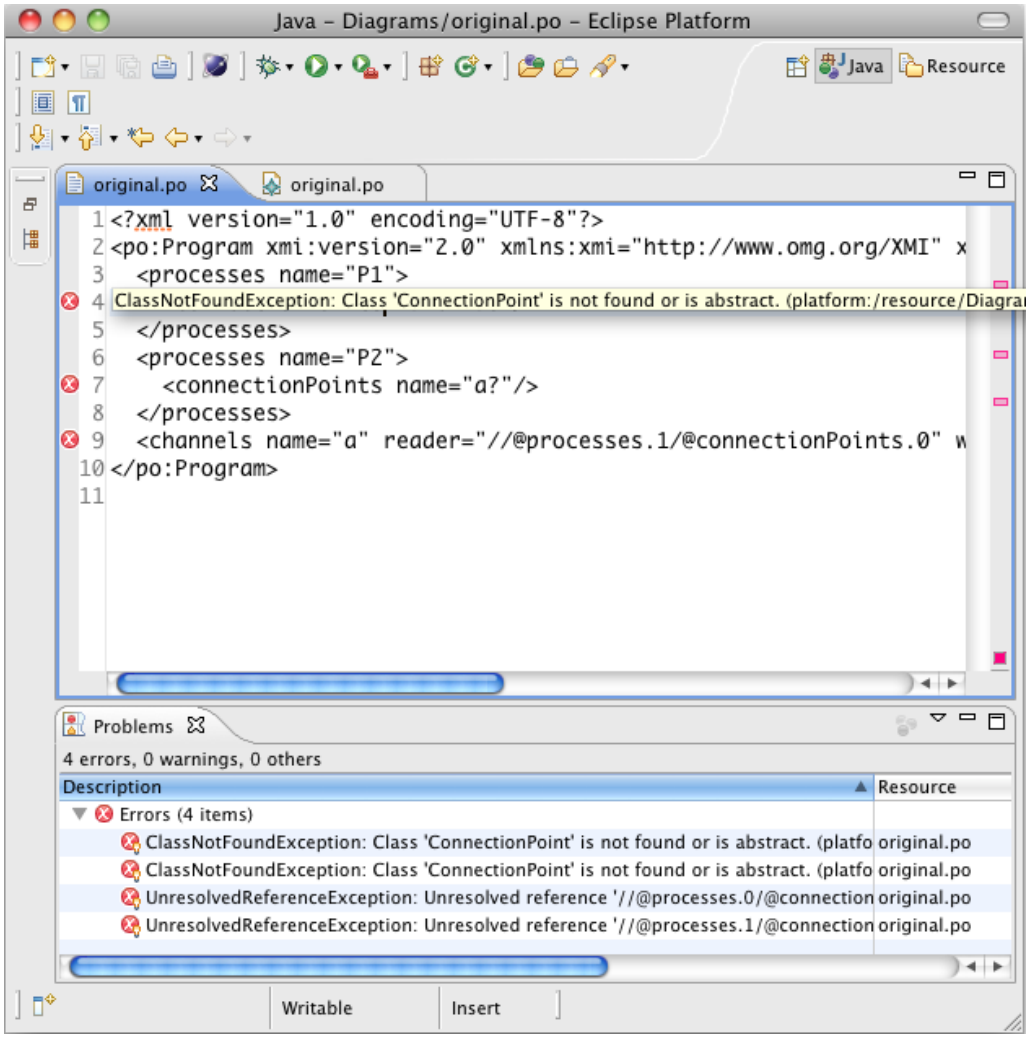


Figure 6.6: XMI prior to migration

model elements. The mistake occurred because XMI identifies model elements using an offset from the root of the document. For example, consider the XMI shown in Figure 6.7. The channel on line 9 specifies the value "//@processes.1/@connectionPoints.0" for its `reader` attribute. The value is an XMI path referencing the first connection point ("@connectionPoints.0") contained in the second process ("@processes.1") of this document ("//"); in other words the connection point on line 7. One of Sampson's models contained many channels and connection points and incorrectly counting the connection points in the model led to several mistakes during the manual editing of the XMI.

As demonstrated above, migration using only the tools provided by EMF

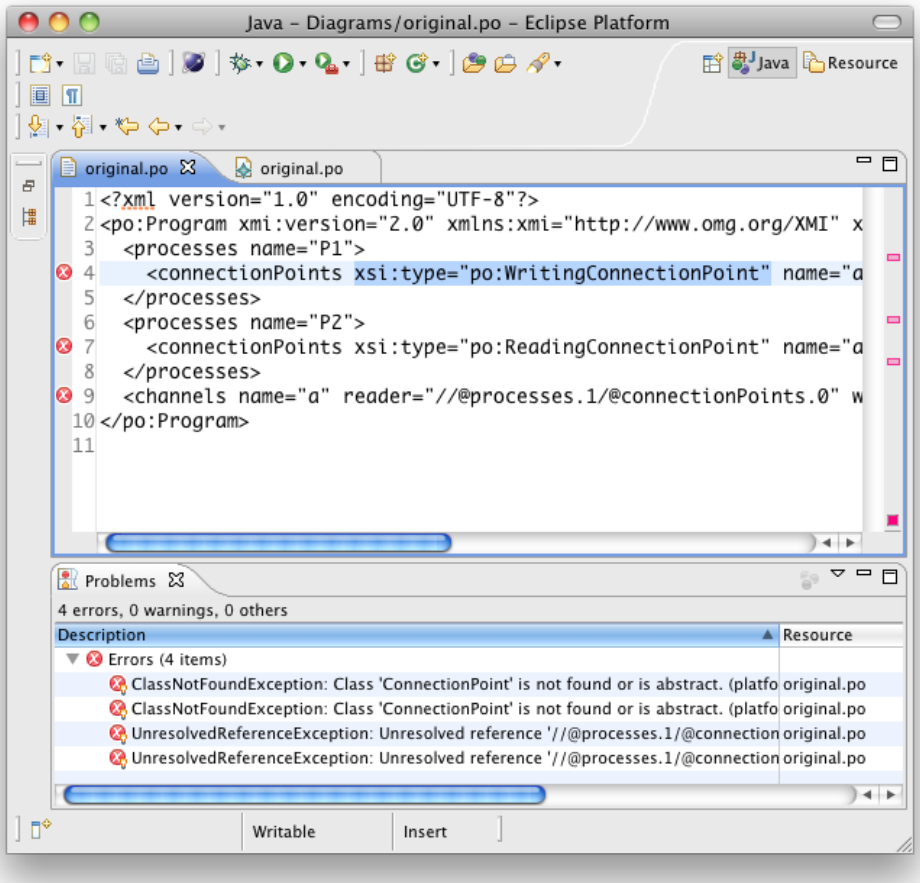


Figure 6.7: XMI after migration

was an iterative process, and mistakes were made. The sequel demonstrates that, using the dedicated structures described in Chapter 5, migration can be performed in one iteration, without requiring the developer to switch between a conformance reporting and reconciling tools. In addition, the sequel suggests how the mistake described above might be avoided by using a textual modelling notation that has been optimised for human-use.

6.1.4 User-Driven Co-Evolution with Dedicated Structures

Chapter 5 describes two structures that can be used to perform user-driven co-evolution. Here, the functionality of the two structures, a metamodel-independent syntax and a textual modelling notation, is summarised. Subsequently, an approach that uses the metamodel-independent syntax and the textual modelling notation for migrating the model from the process-oriented

example is presented. The model migration example presented in this section was performed after the process-oriented editor was completed, and demonstrates how migration might have been achieved with dedicated structures for user-driven co-evolution. The sequel compares the user-driven co-evolution approach presented in this section with the approach presented above.

The metamodel-independent syntax presented in Section 5.1 allows non-conformant models to be loaded, and for the conformance of models to be checked against any metamodel. The textual modelling notation presented in Section 5.2 is built atop the metamodel-independent syntax and provides an alternative to XMI for editing models with a textual representation. Together, the two structures can be used for performing user-driven co-evolution using the approach shown in Figure 6.8. First, the user attempts to load a model. When the model cannot be loaded, the user clicks the "Generate HUTN" menu item, and the model's XMI is automatically transformed to HUTN by the implementation of HUTN described in Chapter 5. The generated HUTN source codes is presented in an editor that automatically reports conformance problems. The user edits the HUTN to reconcile conformance problems. The conformance report is automatically updated while the user edits the HUTN source. When the conformance problems are fixed, XMI for the conformant model is automatically generated, and migration is complete. The model can then be loaded in the graphical editor.

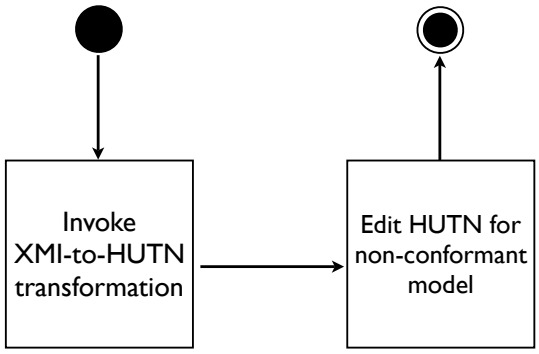


Figure 6.8: User-driven co-evolution with dedicated structures

The way in which the approach shown in Figure 6.8 might have been used for performing user-driven co-evolution for the process-oriented metamodel is now demonstrated. For the model shown in Figure 6.2, the HUTN shown in Figure 6.9 was generated by invoking the automatic XMI-to-HUTN transformation. The HUTN development tools automatically present any conformance problems, as shown in the bottom pane (and the left-hand margin of the top pane) in Figure 6.9.

Conformance problems are reconciled manually by the user, who edits the

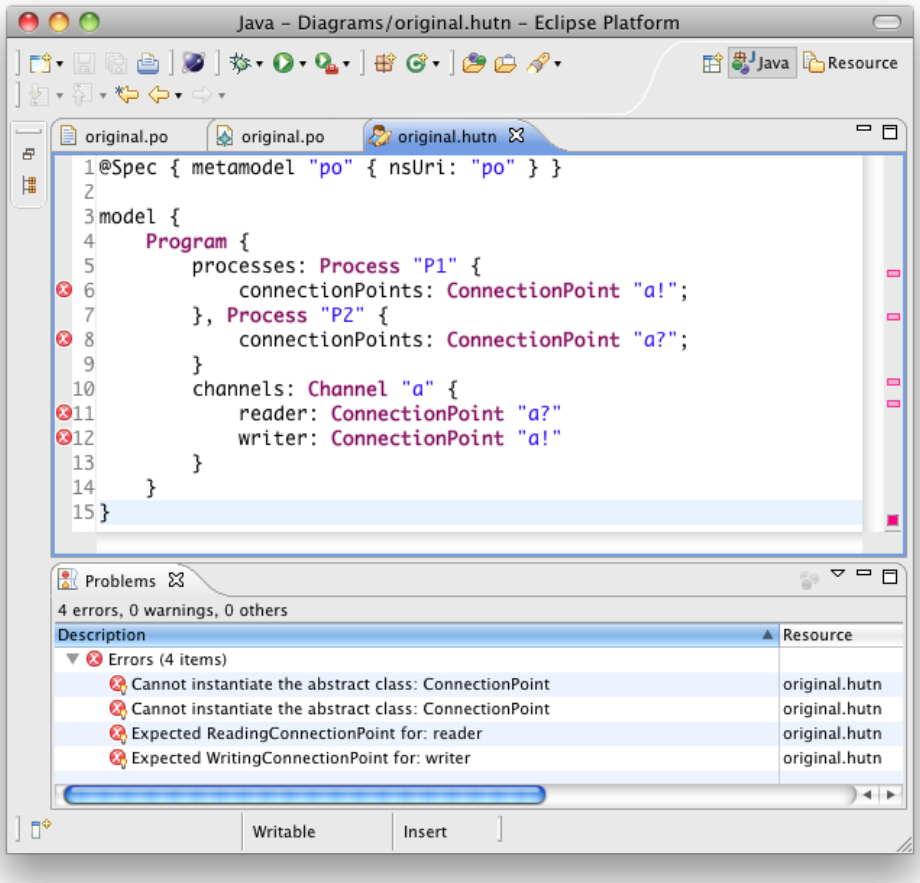


Figure 6.9: HUTN source prior to migration

HUTN source. Conformance is automatically checked whenever the HUTN is changed. For example, Figure 6.10 shows the HUTN editor when migration is partially complete. Some of the conformance problems have been reconciled, and the associated error-markers are no longer displayed in the left-hand margin.

As discussed in Section 6.1.3, mistakes were made when migrating one of Sampson's models, probably because of the way in which XMI specifies reference values. In HUTN, reference values are specified by name. The channel on lines 10-13 of Figure 6.9 refers to its reader and writer by name ("a?" and "a!" respectively), rather than using a path (such as "//@processes.1/@connectionPoints.0") as is the case with XMI.

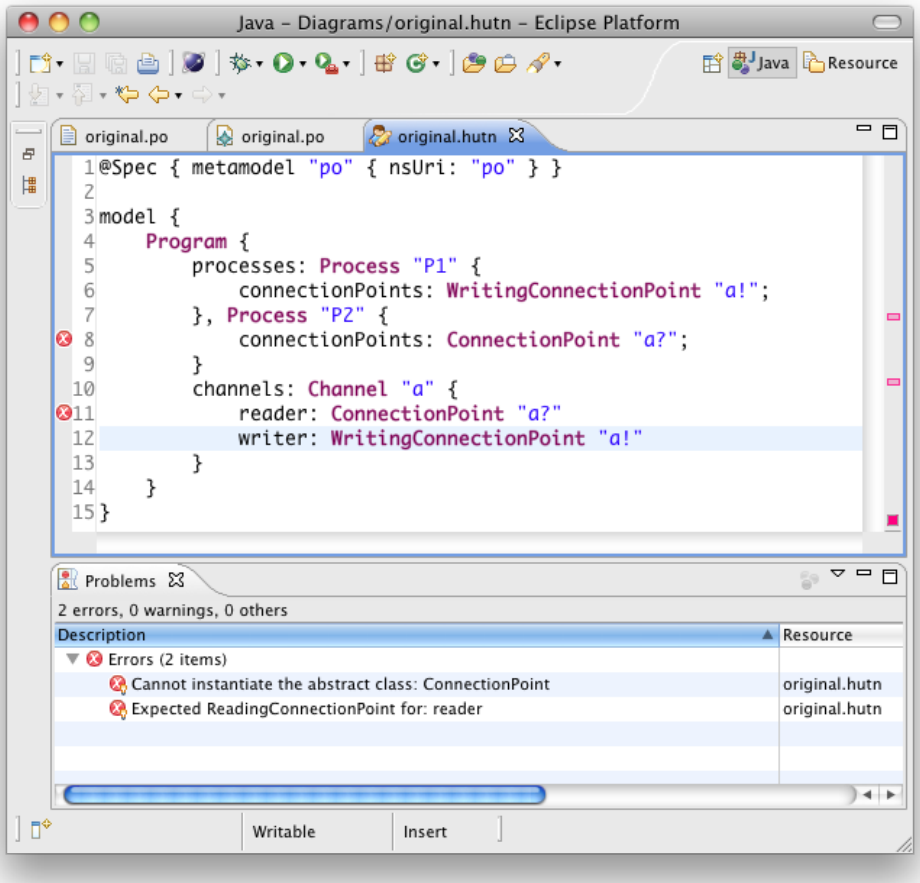


Figure 6.10: HUTN source part way through migration

6.1.5 Comparison

To suggest ways in which dedicated structures for user-driven co-evolution might increase developer productivity, the two user-driven co-evolution approaches demonstrated above are now compared. The first approach, described in Section 6.1.3, uses only those tools available in EMF for performing user-driven co-evolution, while the second approach, described in Section 6.1.4 uses two of the structures described in Chapter 5. Applying the approaches to the process-oriented example highlighted differences between the modelling notations used, and the way in which conformance problems were reported.

Differences in modelling notation

The two approaches used different modelling notations for reconciling conformance problems. The first approach used XMI, while the second used HUTN. The differences in notation that might influence developer productivity during user-driven co-evolution are now discussed. However, further work is required to more rigorously explore the extent to which developer productivity is affected by the modelling notation, as discussed in Section 6.1.6.

The way in which the type of a model element is specified varies between XMI and HUTN. In XMI, type information can be omitted in some circumstances, but must be included in others. In HUTN, type information is mandatory for every model element. Consequently, every HUTN document contains examples of how type information should be specified, whereas XMI documents may not.

Reference values are specified using paths in XMI (such as "//@processes.1/@connectionPoints.0") and by name (such as "a?") in HUTN. XMI paths are constructed in terms of a document's structure and, as such, rely on implementation details. The name of a model element, on the other hand, is specified in the model, and does not rely on any implementation details. Consequently, it is conceivable that fewer mistakes will be made during user-driven co-evolution when reference values are specified by name rather than using an XMI path.

Differences in conformance reports

The two approaches varied in the way in which conformance problems were reported, and, as a consequence, the first approach was iterative and the second was not. The way in which these differences might influence developer productivity during user-driven co-evolution are now discussed. Again, further work is required to more rigorously explore the extent to which developer productivity is affected by the differences in conformance reporting, as discussed in Section 6.1.6.

With EMF, user-driven co-evolution is an iterative process. Conformance errors are fixed by the user, who then reloads the reconciled model (with, for example, a graphical editor). Each time the model is loaded, further conformance problems might be reported when, for example, the user makes a mistake when reconciling the model. By contrast, the implementation of HUTN described in Section 5.2 uses a background compiler that checks conformance while the user edits the HUTN source. When the user makes a mistake reconciling the HUTN source, the error is reported immediately, and does not require the model to be loaded in the graphical editor.

Although not demonstrated in the example considered in this section, user-driven co-evolution would, for some types of metamodel changes, remain an iterative process even if EMF performed conformance checking in the background. Because EMF uses a multi-pass parser, some types of conformance

problem are reported before other types. For example, conformance problems relating to multiplicity constraints (for example, a process does not specify a name, but name is a mandatory attribute) are reported after all other types of conformance problem. When several types of conformance problem have been affected by metamodel changes, user-driven co-evolution with EMF would remain an iterative process. Single-pass, background parsing is required to display all conformance problems while the user migrates a model.

6.1.6 Towards a more thorough comparison

Although the above comparison suggests that dedicated structures for performing user-driven co-evolution might increase developer productivity, further research is required to more rigorously evaluate this claim. The ways in which this evaluation might be extended in the future are now discussed.

A comprehensive user study, involving hundreds of users, is one means for exploring the extent to which productivity varies when dedicated structures are used to perform user-driven co-evolution. Ideally, participants for the study would constitute a large and representative sample of the users of EMF. Productivity might be measured by the time taken to perform co-evolution. To remove a potential source of bias, several examples of co-evolution might be used.

Locating a reasonable number of participants and co-evolution examples for a comprehensive user study was not feasible in the context of this thesis. Nevertheless, the comparison presented in Section 6.1.5 suggests that productivity might be increased when using dedicated structures for user-driven co-evolution. By demonstrating an approach to user-driven co-evolution that uses dedicated structures, this thesis provides a foundation for further, more rigorous evaluation. For example, the HUTN specification [OMG 2004] makes claims about the human-usability of the notation, but the usability of HUTN has not been studied or compared with other modelling notations. The implementations of the co-evolution structures described in Chapter 5 enable further comparisons.

6.1.7 Summary

This section has demonstrated two approaches to user-driven co-evolution using a co-evolution example from a project in which a graphical model editor was created for process-oriented programs. The first approach used the structures available in EMF alone, while the second approach used two of the structures described in Chapter 5. Comparing the two approaches highlighted differences between the way in which conformance problems were reported and between the modelling notations used to reconcile conformance problems. The comparison described in Section 6.1.5 suggests that developer produc-

tivity might be increased by using the second approach, but, as discussed in Section 6.1.6, further work is required to more rigorously evaluate this claim.

6.2 Quantitive Comparison of Model Migration Languages

In Section 4.3, the following research requirement was identified: *This thesis must implement and evaluate a domain-specific language for specifying and executing model migration strategies, comparing it to existing languages for specifying model migration strategies.* As discussed in Section 5.4, this thesis contributes Epsilon Flock, a domain-specific language for model migration. This section approches the second part of the above research requirement by comparing Flock with languages that are used in contemporary migration tools.

In developer-driven migration, a programming language codifies the migration strategy. Because migration involves deriving the migrated model from the original, migration strategies typically access information from the original model and, based on that information, update the migrated model in some way. As such, migration is written in a language with constructs for accessing and updating the original and migrated models. Here, those language constructs are termed *model operations*. Using examples of co-evolution, this section explores the variation in frequency of model operation over different model migration languages, and discusses to what extent the results of this comparison can be used to assess the suitability of the languages considered for model migration.

As discussed in Chapter 5, the languages currently used for model migration vary. Model-to-model transformation languages are used in some migration tools (e.g. [Cicchetti *et al.* 2008, Garcés *et al.* 2009]), and general-purpose languages in others (e.g. [Herrmannsdoerfer *et al.* 2009b, Hussey & Paternostro 2006]). Irrespective of the language used for migration, the way in which a migration tool relates original and migrated model elements falls into one of two categories: new- or existing-target, which were first introduced in Section 5.3.2. In the former, the migrated model is created afresh by the execution of the migration strategy. In the latter, the migrated model is initialised as a copy of the original model and then the migration strategy is executed.

Flock contributes a novel approach for relating original and migrated model elements, termed conservative copy. Conservative copy is a hybrid of new- and existing-target approaches. This section compares new-target, existing-target and conservative copy in the context of model migration. Section 6.2.1 describes the data used in the comparison. The method for the comparison is discussed in Section 6.2.2. Section 6.2.3 identifies model operations for each of the migration languages used in the comparison, and Section 6.2.4 presents and analysis the results.

6.2.1 Data

Five examples of co-evolution were used to compare new-target, existing-target and conservative copy. This section briefly discusses the data used in the comparison.

Co-evolution Examples

To remove one of the possible threats to the validity of the comparison, the examples used were distinct from those used to identify the thesis requirements in Chapter 4. The five examples used in this section are taken from three projects.

Two examples were taken from the *Newsgroup* project, which performs statistical analysis of NNTP newsgroups and was developed by Dimitris Kolovos, a lecturer in this department. One example was taken from *UML* (the Unified Modeling Language), an OMG specification of a language for modelling software systems. Two examples were taken from *GMF* (Graphical Modeling Framework) [Gronback 2009], an Eclipse project for generating graphical model editors.

Selection of Migration Languages

As discussed above, there are two ways in which existing migration languages relate original and migrated model elements, new- and existing-target. Flock contributes a third way, conservative copy. For the comparison with Flock, one new- and one existing-target language was chosen.

The Atlas Transformation Language (ATL), a model-to-model transformation language that has been used in [Cicchetti *et al.* 2008, Garcés *et al.* 2009] for model migration. As discussed in Section 5.3.2, model-to-model transformation languages support only new-target transformations for model migration[1].

The author is aware of two approaches to migration that use existing-target transformations. In COPE [Herrmannsdoerfer *et al.* 2009b], migration strategies can be hand-written in Groovy when no co-evolutionary operator is applicable. As discussed in Section 5.3.2, COPE's Groovy migration strategies use an existing-target approach. COPE provides six functions for interacting with model elements, such as `set`, for changing the value of a feature, and `unset`, for removing all values from a feature. In the remainder of this section, the term *Groovy-for-COPE* is used to refer to the combination of the Groovy programming language and the functions provided by COPE for use in hand-written migration strategies. In Ecore2Ecore [Hussey & Paternostro 2006], migration is performed when the original model is loaded, effectively an existing-target approach.

[1] Because, in model migration, the source and target metamodels are not the same.

The comparison to Flock described in this section uses ATL to represent new-target approaches and Groovy-for-COPE to represent existing-target approaches. Groovy-for-COPE was preferred to Ecore2Ecore because the latter is not as expressive[2] and cannot be used for migration in the co-evolution examples considered here.

6.2.2 Method

For each example of co-evolution, a migration strategy was written using each migration language (ATL, Groovy-for-COPE and Flock). The co-evolution examples considered included reference migrated models. The correctness of each migration strategy was assured by comparing the migrated model produced with the corresponding reference migrated model.

Next, a set of model operations were identified, as described in Section 6.2.3 and a program was written to count the frequency of each model operation in each migration strategy. The counting program was tested by comparison to manual counts.

The method described above introduces one non-trivial threat to the validity of the comparison. Because the author is obviously more familiar with Flock than with ATL and Groovy-for-COPE, it is possible that the migration strategies written in the latter two languages may contain more model operations than necessary. In some cases, it was possible to reduce the effects of this threat by re-using or adapting existing migration strategy code written by the migration language authors. This is discussed further in the sequel.

6.2.3 Model Operations

The way in which model operations were counted is now described. Two categories of model operation were considered: copying and deleting. Copying (deleting) operations are used to assign (remove) values to (from) elements of the migrated model. The three languages considered use different syntax for semantically equivalent statements. Some of the languages provide more than one construct for copying or deleting values. The remainder of this section lists the concrete syntax of the model operations in each language. In addition, the extent to which the comparison described in this section was able to use code written by the authors of each language is discussed.

Atlas Transformation Language (ATL)

For the Atlas Transformation Language (ATL), the following model operations were counted:

[2]Communication with Ed Merks, Eclipse Modeling Project leader, 2009, available at `http://www.eclipse.org/forums/index.php?t=tree&goto=486690&S=b1fdb2853760c9ce6b6b48d3a01b9aac`

- **Copying:** Assignment to a feature:

```
<feature> <- <value>
```

Deleting operations are not used in new-target migration strategies. A new-target migration strategy specifies only those values that must appear in the migrated model and, unlike existing-target approaches and conservative copy, no values are copied automatically prior to the execution of the migration.

Groovy-for-COPE

For Groovy-for-COPE, the following model operations were counted:

- **Copying:** Assignment to a feature:

```
<element>.<feature> = <value>
<element>.<feature>.add(<value>)
<element>.<feature>.addAll(<collection_of_values>)
<element>.set(<feature>) = <value>
```

- **Deleting:** Unsetting a feature:

```
<element>.<feature>.unset()
```

- **Deleting:** Removing a model element:

```
delete <element>
```

Groovy-for-COPE provides four assignment statements. The first is typical. The second and third are used to add values to a collection. The fourth is used for reflective access to a model.

The deleting operations are necessary for some existing-target migration strategies, because the migrated model (which is initialised as a copy of the original model) may contain data that is no longer captured in the evolved metamodel.

As discussed in Section 4.2.3, COPE provides a library of built-in, reusable co-evolutionary operators. Each co-evolutionary operator specifies a metamodel evolution along with a corresponding model migration strategy. To mitigate the chance that the author, who is less familiar with COPE than Flock, introduced additional model operations, writing the Groovy-for-COPE migration strategies involved, where possible, applying an appropriate COPE co-evolutionary operator and counting the number of model operations in the generated migration strategy. Not all of the examples considered could be specified using co-evolutionary operators, and, in these cases, the Groovy-for-COPE migration strategy was written entirely by the author.

Epsilon Flock

Epsilon Flock, a transformation language tailored for model migration, was developed in this thesis and discussed in Chapter 5. Flock uses the Epsilon Object Language (EOL) [Kolovos *et al.* 2006] to access and update model values. For Flock, the following model operations were counted:

- **Copying:** Assignment to a feature:

```
<element>.<feature> := <value>
<element>.<feature>.add(<value>)
<element>.<feature>.addAll(<collection_of_values>)
```

- **Deleting:** Removing a model element:

```
delete <element>
```

Like Groovy-for-COPE, Flock provides several assignment statements. The first is typical. The second and third are used to add values to a collection.

Flock provides a remove operation but not an unset. The former is required to remove model elements that no longer conform to the target metamodel. Flock does not provide an unset operation because the conservative copy algorithm will never copy to the migrated model any value that does not conform to the evolved metamodel.

6.2.4 Results

By measuring the number of model operations in model migration strategies, the way in which each co-evolution approach relates original and migrated model elements was investigated. The five examples of model migration discussed above were measured to obtain the results shown in Table 6.1.

In addition, the results from measuring the examples identified from Chapter 4 are shown in Table 6.2. However, because the examples used to produce the measurements shown in Table 6.2 were used to design Flock, the measurements in Table 6.2 are less relevant to the evaluation presented here than the measurements shown in Table 6.1. Nevertheless, the measurements made in Table 6.2 are included in the interest of transparency.

For four of the five examples in Table 6.1, conservative copy requires less model operations than new-target and existing-target. The GMF Graph example is the exception to this trend.

For all of the five examples in Table 6.1, no migration strategy specified with existing-target contained less model operations when encoded with new-target. However, three of the Refactor examples in Table 6.2 do not follow this trend.

The results are now investigated, starting by discussing the differences between the source-target relationships. Investigating the results led to the

	Migration Language Source-Target Relationship		
(Project) Example	**ATL** New	**G-f-C** Existing	**Flock** Conservative
(Newsgroup) Extract Person	9	6	5
(Newsgroup) Resolve Replies	8	3	2
(UML) Activity Diagrams	15	15	8
(GMF) Graph	101	11	14
(GMF) Gen2009	310	16	16

Table 6.1: Model operation frequency (evaluation examples).

	Migration Language Source-Target Relationship		
(Project) Example	**ATL** New	**G-f-C** Existing	**Flock** Conservative
(FPTC) Connections	6	6	3
(FPTC) Fault Sets	7	5	3
(GADIN) Enum to Classes	4	1	0
(GADIN) Partition Cont	5	3	2
(Literature) PetriNets	12	10	6
(Newsgroup) Extract Person	9	6	5
(Newsgroup) Resolve Replies	8	3	2
(Process-Oriented) Split CP	8	1	1
(Refactor) Cont to Ref	4	5	3
(Refactor) Ref to Cont	3	4	3
(Refactor) Extract Class	5	4	2
(Refactor) Extract Subclass	6	0	0
(Refactor) Inline Class	4	5	2
(Refactor) Move Feature	6	2	1
(Refactor) Push Down Feature	6	0	0

Table 6.2: Model operation frequency (analysis examples).

discovery of two limitations of the conservative copy implementation in Flock, relating to sub-typing and side-effects during initialisation. These limitations are also discussed below.

Source-Target Relationships

New-target, existing-target and conservative copy initialise the migrated model in a different way. New-target initialises an empty model, while existing-target initialises a complete copy of the original model. Conservative copy initialises the migrated model by copying only those model elements from the original model that conform to the migrated metamodel.

New- and existing-target are opposites. In the former, explicit assignment operations must be used to copy values from original to migrated model for each feature that is not affected by the metamodel evolution. By contrast, in the latter unset operations must be used when the value of a feature should not have been copied.

In situations where a large number of metamodel features have not been affected by evolution, expressing migration with a new-target transformation language requires more model operations than using an existing-target transformation language. This is particularly noticeable in the GMF examples shown in Table 6.1, where ATL (new-target) requires many more model operations than Groovy-for-COPE (existing-target) and Flock (conservative copy).

In situations where a large number of metamodel features have been renamed, expressing migration with an existing-target transformation language requires more model operations than using a new-target transformation language. This is because, in an existing-target transformation language, two model operations (an unset and an assignment) are needed to migrate values in response to the renaming of a feature:

```
<element>.<newFeature> = <element>.unset(<oldFeature>)
```

By contrast, a new-target transformation language requires only one model operation (an assignment):

```
<migrated_element>.<feature> = <original_element>.<feature>
```

The UML (Table 6.1) and Refactor Inline Class (Table 6.2) examples contained several feature renamings, and consequently the Groovy-for-COPE (existing-target) figure was nearer to the ATL (new-target) figure than the Flock (conservative copy) figure. This is contrary to the trend in Tables 6.1 and 6.2.

Conservative copy is a hybrid of new- and existing target. Model values that have been affected by evolution are not copied to the migrated model, and so the migration strategy need not unset affected model values. Model values that have not been affected by evolution are copied to the migrated model, and so the migration strategy need not explicitly copy unaffected model values.

Two conclusions can be drawn from this discussion. Firstly, in general, less model operations are used when specifying a migration strategy with a con-

servative copy migration language than when specifying the same migration strategy with a new- or existing-target migration language.

Secondly, in the examples studied here, there are often more features unaffected by metamodel evolution than affected. Consequently, specifying model migration with a new-target migration language requires more model operations than in an existing-target migration language for the examples shown in Tables 6.1 and 6.2. [Sprinkle 2003] proposes that metamodel evolution often involves changes to relatively few metamodel elements, and the results presented in this section support his argument.

Subtyping

Form the GMF Graph example (Table 6.1), conservative copy requires more model operations than existing-target. Investigation of this result revealed a limitation in conservative copy limitation in Flock, which is now described.

Figure 6.11 shows a simplified part of the GMF Graph metamodel prior to evolution. When the metamodel evolved, the types of the `figure` and `accessor` features were changed. Consequently, the migration strategy needed to change the values stored in the `figure` and `accessor` features. In the simplified example presented here, the types of the `figure` and `accessor` features were changed from string to integer. The intended migration semantics are for the integer value to be the length of the original string value. This is a simplification that is representative of the actual GMF Graph metamodel evolution.

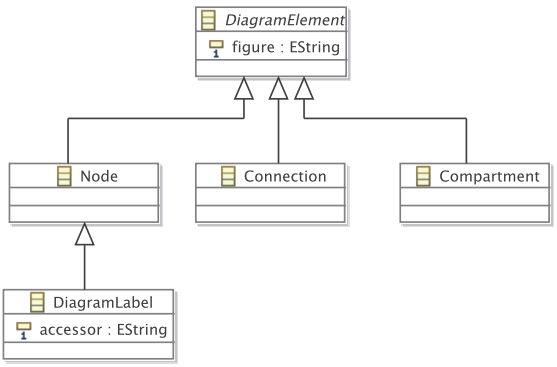


Figure 6.11: Simplified fragment of the GMF Graph metamodel.

In ATL, the migration strategy for the metamodel evolution discussed above can be expressed using two model operations, because an ATL transformation rule may inherit the body of another. The `DiagramElements` rule on lines 1-4 of Listing 6.1 specifies that the value of the figure feature should be the length of the original value. For `Nodes`, `Connections` and `Compartments`, migration can be specified simply by extending the `DiagramElements` rule

(lines 6-19). For DiagramLabels, the values of both the accessor and figure feature must be migrated. On lines 21-24, the DiagramLabels extends Nodes and hence DiagramElements to inherit the body of the latter for migrating figures, and, in addition, the DiagramLabels rule defines the migration for the value of the accessor feature.

```
1  abstract rule DiagramElements {
2   from o : Before!DiagramElement
3   to  m : After!DiagramElement ( figure <- o.figure.length() )
4  }
5
6  rule Nodes extends DiagramElements {
7   from o : Before!Node
8   to  m : After!Node
9  }
10
11 rule Connections extends DiagramElements {
12  from o : Before!Connection
13  to  m : After!Connection
14 }
15
16 rule Compartments extends DiagramElements {
17  from o : Before!Compartment
18  to  m : After!Compartment
19 }
20
21 rule DiagramLabels extends Nodes {
22  from o : Before!DiagramLabel
23  to  m : After!DiagramLabel ( accessor <- o.accessor.length() )
24 }
```

Listing 6.1: Simplified GMF Graph model migration in ATL

In Groovy-for-COPE, the migration is similar to ATL. However, Groovy-for-COPE is entirely imperative, and so the migration (Listing 6.2) is more concise than the ATL migration (Listing 6.1). In Listing 6.2, a loop iterates over each instance of DiagramElement (line 1), migrating the value of its figure feature (line 2). If the DiagramElement is also a DiagramLabel (line 4), the value of its accessor feature is also migrated (line 5).

```
1  for (diagramElement in subtyping.DiagramElement.allInstances())
       {
2   diagramElement.figure = diagramElement.figure.length()
3
```

```
4   if (subtyping.DiagramLabel.allInstances.contains(
        diagramElement)) {
5    diagramElement.accessor = diagramElement.accessor.length()
6   }
7 }
```

Listing 6.2: Simplified GMF Graph model migration in COPE

In both ATL and Groovy-for-COPE, only 2 model operations are required for this migration: an assignment for each of the two features being migrated. However, the equivalent Flock migration strategy, shown in Listing 6.3, requires 5 model operations. In Flock, a migrate rule must be specified for each concrete subtype of `DiagramElement`. A single `DiagramElement` rule cannot be used to migrate the concrete subtypes because, when a rule does not specify a `to` part, Flock will create an instance of the type named after the `migrate` keyword (i.e. `DiagramElement` here). Because `DiagramElement` is abstract, Flock will fail with a runtime error. Furthermore, because only one rule can be applied to each original model element, the `DiagramLabel` rule (lines 9-12) must migrate the values of both the figure and accessor features, and cannot exploit the kind of re-use provided by ATL with rule inheritance.

```
1  migrate Compartment {
2   migrated.figure := original.figure.length();
3  }
4
5  migrate Connection {
6   migrated.figure := original.figure.length();
7  }
8
9  migrate DiagramLabel {
10  migrated.figure := original.figure.length();
11  migrated.accessor := original.accessor.length();
12 }
13
14 migrate Node {
15  migrated.figure := original.figure.length();
16 }
```

Listing 6.3: Simplified GMF Graph model migration in Flock

The example presented in this section highlights a limitation of the conservative copy algorithm as it is implemented in Flock. The extent to which this limitation can be addressed with changes to the implementation of Flock, and in general, is discussed in Section 7.2. This section now considers one further limitation of existing-target and conservative copy migration languages.

Side-Effects during Initialisation

The measurements observed for one of the examples of co-evolution from Chapter 4, Change Reference to Containment, cannot be explained by the conceptual differences between source-target relationship. Instead, the way in which the source-target relationship is implemented must be considered.

When a reference feature is changed to a containment reference during metamodel evolution, constructing the migrated model by starting from the original model (as is the case with existing-target and conservative copy) can have side-effects which complicate migration.

In the Change Reference to Containment example, a `System` initially comprises `Ports` and `Signatures` (Figure 6.12). A `Signature` references any number of `ports`. The metamodel is to be evolved so that `Ports` can no longer be shared between `Signatures`.

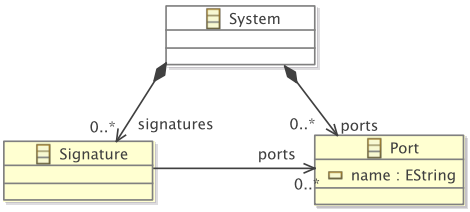


Figure 6.12: Original metamodel.

The evolved metamodel is shown in Figure 6.13. `Signatures` now contain - rather than reference - `Ports`. Consequently, the `ports` feature of `System` is no longer required and is removed.

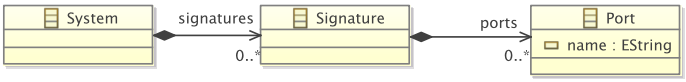


Figure 6.13: Evolved metamodel.

The migration strategy is straightforward in a new-target migration language: for each `Signature` in the original model, each member of the `ports` feature is cloned and added to the `ports` feature of the equivalent `Signature`.

```
1 rule Systems {
2   from
3     o : Before!System
4   to
5     m : After!System ( signatures <- o.signatures )
6 }
```

```
7
8  rule Signature {
9    from
10     o : Before!Signature
11   to
12     m : After!Signature (
13       ports <- o.ports->collect(p | thisModule.Port(p))
14     )
15 }
16
17 lazy rule Port {
18   from
19     o : Before!Port
20   to
21     m : After!Port ( name <- o.name )
```

Listing 6.4: Change R to C model migration in ATL

In existing-target and conservative copy migration languages, migration is less straightforward because the value of a containment reference (`Signature#ports`) is set automatically by the migration strategy execution engine. When a containment reference is set, the contained objects are removed from their previous containment reference (i.e. setting a containment reference can have side-effects). Therefore, in a `System` where more than one `Signature` references the same `Port`, the migrated model cannot be formed by copying the contents of `Signature#ports` from the original model. Attempting to do so causes each `Port` to be contained only in the last referencing `Signature` that was copied.

In COPE, conformance is checked only after execution of the migration strategy, when the model is transformed to a metamodel-specific representation. Therefore, the containment nature of the reference is not enforced until after the migration strategy is executed. Hence, the migration strategy discussed here can be specified by unsetting the contents of the `ports` reference (line 4 of Listing 6.5), and creating a copy of each referenced `Port` (lines 5-7 of Listing 6.5).

Unlike the ATL migration strategy, the ports in the Groovy-for-COPE migration strategy are cloned in the same model as the original port. Consequently, the Groovy-for-COPE migration strategy must either only clone ports that are referenced by more than one signature or clone every referenced port, but delete all of the original ports. The latter approach requires 2 more model operations (to populate and delete the original ports) than the former (shown in Listing 6.5).

```
1 def contained = []
2
```

```
3  for(signature in refactorings_changeRefToCont.Signature.
       allInstances) {
4   for(port in signature.ports)) {
5    // when more than one Signature references this port
6    if (contained.contains(port)) {
7     def clone = Port.newInstance()
8     clone.name = port.name
9     signature.ports.add(clone)
10    signature.ports.remove(port)
11   } else {
12    contained.add(port)
13   }
14  }
15 }
16
17 for(port in refactorings_changeRefToCont.Port.allInstances) {
18  if (not refactorings_changeRefToCont.Signature.allInstances.
       any { it.ports.contains(port) }) {
19    port.delete()
20  }
21 }
```

Listing 6.5: Change R to C model migration in COPE

In Flock, the containment nature of the reference is enforced when the migrated model is initialised. Because changing the contents of a containment reference can have side-effects, a `Port` that appears in the `ports` reference of a `Signature` in the original model may not have been automatically copied to the `ports` reference of the equivalent `Signature` in the migrated model during initialisation. Consequently, the migration strategy must check the `ports` reference of each migrated `Signature`, cloning only those `Ports` that have not be automatically copied during initialisation (see line 3 of Listing 6.6).

```
1  migrate Signature {
2   for (port in original.ports) {
3    if (migrated.ports.excludes(port.equivalent())) {
4     var clone := new Migrated!Port;
5     clone.name := port.name;
6     migrated.ports.add(clone);
7    }
8   }
9  }
10
11 delete Port when: not Original!Signature.all.exists(s|s.ports.
```

```
includes(original))
```

Listing 6.6: Change R to C model migration in Flock

The Groovy-for-COPE and Flock migration strategies must also remove any `Ports` which are not referenced by any `Signature` (lines 17-21 of Listing 6.5, and line 11 of Listing 6.6 respectively), whereas the ATL migration strategy, which initialises any empty migrated model, does not copy unreferenced `Ports`.

When a non-containment reference is changed to a containment reference, producing a migration strategy in Flock and Groovy-for-COPE requires the user to be aware of the side-effects that can occur during initialisation. It may be possible to extend the existing-target and conservative copy algorithms used in COPE and Flock to automatically perform cloning when a reference is changed to be a containment reference. This is discussed further in Section 7.2.

6.2.5 Summary

By measuring frequency of model operations, this section has compared, in the context of model migration, three approaches to relating source-target relationship: new-target, existing-target and conservative copy. The results have been analysed and the measurement method described.

The analysis of the measurements has shown that new- and existing-target migration languages are better suited to different contexts. New-target languages require less model operations than existing-target languages when metamodel evolution involves the renaming of features. Existing-target languages require less model operations than new-target languages when metamodel evolution does not affect most model elements. For the examples considered here, the latter context was more common. Conservative copy requires less model operations than both new- and existing-target in almost all of the examples considered here.

The comparison has highlighted two limitations of the conservative copy algorithm implemented in Epsilon Flock, and this section has shown how these limitations are problematic for specifying some types of migration strategy.

The author is not aware of any existing quantitive comparisons of migration languages, and, as such, the best practices for conducting such comparisons are not clear. The method used in obtaining these measurements has been described to provide a foundation for future comparisons.

6.3 Migration Tool Comparison

As discussed in Chapter 3, several tools for managing co-evolution are described in the literature. Chapter 5 proposes a further tool, Epsilon Flock. While each tool has strengths and weaknesses, little is known about how migration tools compare in practice, which makes tool selection more challenging.

Consequently. Chapter 4 identified the following thesis requirements: *This thesis must compare and evaluate existing languages for specifying model migration strategies. This thesis must implement and evaluate a domain-specific language for specifying and executing model migration strategies,* ***comparing it to existing languages for specifying model migration strategies****.*

To approach these two requirements, this section describes work that compares Flock and three existing model migration that were selected from those described in Chapter 4. Following the process outlined in Section 6.3.1, the tools were applied to two co-evolution examples to facilitate their comparison. To improve the validity of the comparison, the developers of each tool were invited to participate. The remainder of this section reports our experiences with each tool (Section 6.3.2), and synthesises advice and guidelines for identifying the most appropriate model migration tool in different situations (Section 6.3.3).

This section is based on joint work with Markus Herrmannsdöerfer (a research student at Technische Universität München), James Williams (a research student in this department), Dimitrios Kolovos (a lecturer in this department) and Kelly Garcés (a research student at EMN-INRIA / LINA-INRIA in Nantes), and has been published in [Rose *et al.* 2010a]. Garcés provided assistance with installing and configuration one of the migration tools, and commented on a draft of the paper. Herrmannsdöerfer, Williams and Kolovos played a larger role in the comparison. Here, the work is narrated to make clear their contributions.

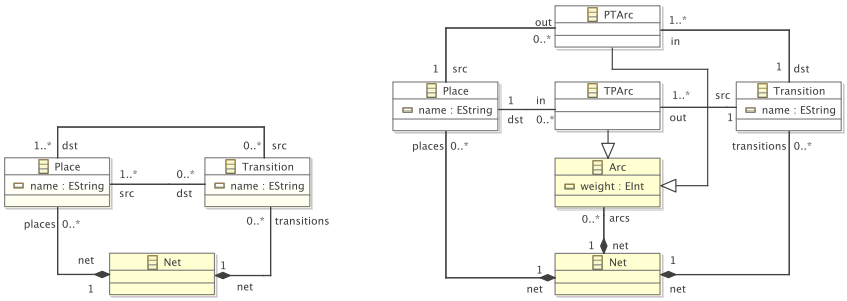
6.3.1 Comparison Method

The comparison described in this section is based on practical application of the tools to the co-evolution examples described below. This section also discusses the tool selection and comparison processes. Herrmannsdöerfer and I identified the co-evolution examples, and formulated the comparison process.

Co-Evolution Examples

To compare migration tools, two examples of co-evolution were used. The first is a well-known problem in the model migration literature and was used to test the installation and configuration of the migration tools, as discussed in Section 6.3.1. The second is a larger example taken from a real-world model-driven development project, and was identified as a potentially useful example for co-evolution case studies in Chapter 4 and in [Herrmannsdoerfer *et al.* 2009a].

Petri Nets. The first example is an evolution of a Petri net metamodel, previously used to describe the implementation of Epsilon Flock (Section 5.4), and in [Cicchetti *et al.* 2008, Garcés *et al.* 2009, Wachsmuth 2007] to discuss co-evolution and model migration.



(a) Original metamodel. (b) Evolved metamodel.

Figure 6.14: Petri nets metamodel evolution (taken from [Rose *et al.* 2010e]). Shading is irrelevant.

In Figure 6.14(a), a Petri `Net` comprises `Places` and `Transitions`. A `Place` has any number of `src` or `dst` `Transitions`. Similarly, a `Transition` has at least one `src` and `dst` `Place`. In this example, the metamodel in Figure 6.14(a) is to be evolved to support weighted connections between `Places` and `Transitions` and between `Transitions` and `Places`.

The evolved metamodel is shown in Figure 6.14(b). `Places` are connected to `Transitions` via instances of `PTArc`. Likewise, `Transitions` are connected to `Places` via `TPArc`. Both `PTArc` and `TPArc` inherit from `Arc`, and therefore can be used to specify a `weight`.

GMF. The second example is taken from the Graphical Modeling Framework (GMF) [Gronback 2009], an Eclipse project for generating graphical editors for models. The development of GMF is model-driven and utilises four domain-specific metamodels. Here, we consider one of those metamodels, GMF Graph, and its evolution between GMF versions 1.0 and 2.0.

The GMF Graph metamodel (not illustrated) describes the appearance of the generated graphical model editor. The metaclasses `Canvas`, `Figure`, `Node`, `DiagramLabel`, `Connection`, and `Compartment` are used to represent components of the graphical model editor to be generated. The evolution in the GMF Graph metamodel was driven by analysing the usage of the `Figure#referencingElements` reference, which relates `Figures` to the `DiagramElements` that use them. As described in the GMF Graph documentation[3], the `referencingElements` reference increased the effort required to re-use figures, a common activity for users of GMF. Furthermore, `referencingElements` was used only by the GMF code generator to determine whether an accessor should be generated for nested `Figures`.

[3] `http://wiki.eclipse.org/GMFGraph_Hints`

In GMF 2.0, the Graph metamodel was evolved to make re-using figures more straightforward by introducing a proxy [Gamma *et al.* 1995] for `Figure`, termed `FigureDescriptor`. The original `referencingElements` reference was removed, and an extra metaclass, `ChildAccess`, was added to make more explicit the original purpose of `referencingElements` (accessing nested `Figures`).

GMF provides a migrating algorithm that produces a model conforming to the evolved Graph metamodel from a model conforming to the original Graph metamodel. In GMF, migration is implemented using Java. The GMF source code includes two example editors, for which the source code management system contains versions conforming to GMF 1.0 and GMF 2.0. For the comparison of migration tools described in this paper, the migrating algorithm and example editors provided by GMF were used to determine the correctness of the migration strategies produced by using each model migration tool.

Compared Tools

The comparison described in this section included one tool from each of the three categories identified in Chapter 4 – *manual specification*, *operator-based* and *metamodel matching* approaches. The tools selected were Epsilon Flock, COPE [Herrmannsdoerfer *et al.* 2009b] and the AtlanMod Matching Language (AML) [Garcés *et al.* 2009], respectively. A further tool from the manual specification category, Ecore2Ecore, was included because it is distributed with the Eclipse Modeling Framework (EMF), arguably the most widely used modelling framework. AML, COPE and Ecore2Ecore were discussed in Chapter 4, and Epsilon Flock in Chapter 5.

Comparison Process

The comparison of migration tools was conducted by applying each of the four tools (Ecore2Ecore, AML, COPE and Flock) to the two examples of co-evolution (Petri nets and GMF). The developers of each tool were invited to participate in the comparison. The authors of COPE and Flock were able to participate fully, while the authors of Ecore2Ecore and AML were available for guidance, advice, and to comment on preliminary results.

Each tool developer was assigned a migration tool to apply to the two co-evolution examples. Because the authors of Ecore2Ecore and AML were not able to participate fully in the comparison, two colleagues experienced in model transformation and migration, James Williams and Dimitrios Kolovos, stood in. To improve the validity of the comparison, each tool was used by someone other than its developer. Other than this restriction, the tools were allocated arbitrarily.

The comparison was conducted in three phases. In the first phase, criteria against which the tools would be compared were identified by discussion

between the tool developers. In the second phase, the first example of co-evolution (Petri nets) was used for familiarisation with the migration tools and to assess the suitability of the comparison criteria. In the third phase, the tools were applied to the larger example of co-evolution (GMF) and results were drawn from the experiences of the tool developers. Table 6.3 summarises the comparison criteria used, which provide a foundation for future comparisons. The next section presents, for each criterion, observations from applying the migration tools to the co-evolution examples.

Table 6.3: Summary of comparison criteria.

Name	Description
Construction	Ways in which tool supports the development of migration strategies
Change	Ways in which tool supports change to migration strategies
Extensibility	Extent to which user-defined extensions are supported
Re-use	Mechanisms for re-using migration patterns and logic
Conciseness	Size of migration strategies produced with tool
Clarity	Understandability of migration strategies produced with tool
Expressiveness	Extent to which migration problems can be codified with tool
Interoperability	Technical dependencies and procedural assumptions of tool
Performance	Time taken to execute migration

6.3.2 Comparison Results

This section reports the similarities and differences of each tool, using the nine criteria described above. The migration strategies formulated with each tool are available online[4].

Each subsection below considers one criterion. This section reports the experiences of the developer to which each tool was allocated. As such, this section contains the work of others. Specifically, Herrmannsdöerfer wrote about Epsilon Flock, Williams wrote about COPE and Kolovos wrote about Ecore2Ecore. (I wrote about AML, and the introductions to each criterion).

Constructing the migration strategy

Facilitating the specification and execution of migration strategies is the primary function of model migration tools. This section reports the process for and challenges faced in constructing migration strategies with each tool.

AML. An AML user specifies a combination of match heuristics from which AML infers a migrating transformation by comparing original and evolved metamodels. Matching strategies are written in a textual syntax, which

[4] `http://github.com/louismrose/migration_comparison`

AML compiles to produce an executable workflow. The workflow is invoked to generate the migrating transformation, codified in the Atlas Transformation Language (ATL) [Jouault & Kurtev 2005]. Devising correct matching strategies was difficult, as AML lacks documentation that describes the input, output and effects of each heuristic. Papers describing AML (such as [Garcés *et al.* 2009]) discuss each heuristic, but mostly in a high-level manner. A semantically invalid combination of heuristics can cause a runtime error, while an incorrect combination results in the generation of an incorrect migration transformation. However, once a matching strategy is specified, it can be re-used for similar cases of metamodel evolution. To devise the matching strategies used in this paper, AML's author provided considerable guidance.

COPE. A COPE user applies *coupled operations* to the original metamodel to form the evolved metamodel. Each coupled operation specifies a metamodel evolution along with a corresponding fragment of the model migration strategy. A history of applied operations is later used to generate a complete migration strategy. As COPE is meant for co-evolution of models and metamodels, reverse engineering a large metamodel can be difficult. Determining which sequence of operations will produce a correct migration is not always straightforward. To aid the user, COPE allows operations to be undone. To help with the migration process, COPE offers the *Convergence View* which utilises EMF Compare to display the differences between two metamodels. While this was useful, it can, understandably, only provide a list of explicit differences and not the semantics of a metamodel change. Consequently, reverse-engineering a large and unfamiliar metamodel is challenging, and migration for the GMF Graph example could only be completed with considerable guidance from the author of COPE.

Ecore2Ecore. In Ecore2Ecore model migration is specified in two steps. In the first step, a graphical mapping editor is used to construct a model that declares basic migrations. In this step only very simple migrations such as class and feature renaming can be declared. In the next step, the developer needs to use Java to specify a customised parser (resource handler, in EMF terminology) that can parse models that conform to the original metamodel and migrate them so that they conform to the new metamodel. This customised parser exploits the basic migration information specified in the first step and delegates any changes that it cannot recognise to a particular Java method in the parser for the developer to handle. Handling such changes is tedious as the developer is only provided with the string contents of the unrecognised features and then needs to use low-level techniques – such as data-type checking and conversion, string splitting and concatenation – to address them. Here it is worth mentioning that Ecore2Ecore cannot handle all migration scenarios

and is limited to cases where only a certain degree of structural change has been introduced between the original and the evolved metamodel. For cases which Ecore2Ecore cannot handle, developers need to specify a custom parser without any support for automated element copying.

Flock. In Flock, model migration is specified manually. Flock automatically copies only those model elements which still conform to the evolved metamodel. Hence, the user specifies migration only for model elements which no longer conform to the evolved metamodel. Due to the automatic copying algorithm, an empty Flock migration strategy always yields a model conforming to the evolved metamodel. Consequently, a user typically starts with an empty migration strategy and iteratively refines it to migrate non-conforming elements. However, there is no support to ensure that all non-conforming elements are migrated. In the GMF Graph example, completeness could only be ensured by testing with numerous models. Using this method, a migration strategy can be easily encoded for the Petri net example. For the GMF Graph example whose metamodels are larger, it was more difficult, since there is no tool support for analysing the changes between original and evolved metamodel.

Changing the migration strategy

Migration strategies can change in at least two ways. Firstly, as a migration strategy is developed, testing might reveal errors which need to be corrected. Secondly, further metamodel changes might require changes to an existing migration strategy.

AML. Because AML automatically generates migrating transformations, changing the transformation, for example after discovering an error in the matching strategy, is trivial. To migrate models over several versions of a metamodel at once, the migrating transformations generated by AML can be composed by the user. AML provides no tool support for composing transformations.

COPE. As mentioned previously, COPE provides an undo feature, meaning that any incorrect migrations can be easily fixed. COPE stores a history of *releases* – a set of operations that has been applied between versions of the metamodel. Because the migration code generated from the release history can migrate models conforming to any previous metamodel release, COPE provides a comprehensive means for chaining migration strategies.

Ecore2Ecore. Migrations specified using Ecore2Ecore can be modified via the graphical mapping editor and the Java code in the custom model parser. Therefore, developers can use the features of the Eclipse Java IDE to modify

and debug migrations. Ecore2Ecore provides no tool support for composing migrations, but composition can be achieved by modifying the resource handler.

Flock. There is comprehensive support for fixing errors. A migration strategy can easily be re-executed using a launch configuration, and migration errors are linked to the line in the migration strategy that caused the error to occur. If the metamodel is further evolved, the original migration strategy has to be extended, since there is no explicit support to chain migration strategies. The full migration strategy may need to be read to know where to extend it.

Extensibility

The fundamental constructs used for specifying migration in COPE and AML (operators and match heuristics, respectively) are extensible. Flock and Ecore2Ecore use a more imperative (rather than declarative) approach, and as such do not provide extensible constructs.

AML. An AML user can specify additional matching heuristics. This requires understanding of AML's domain-specific language for manipulating the data structures from which migrating transformations are generated.

COPE provides the user with a large number of operations. If there is no applicable operation, a COPE user can write their own operations using an in-place transformation language embedded into Groovy[5].

Re-use

Each migration tool capture patterns that commonly occur in model migration. This section considers the extent to which the patterns captured by each tool facilitate re-use between migration strategies.

AML. Once a matching strategy is specified, it can potentially be re-used for further cases of metamodel evolution. Match heuristics provide a re-usable and extensible mechanism for capturing metamodel change and model migration patterns.

COPE. An operation in COPE represents a commonly occurring pattern in metamodel migration. Each operation captures the metamodel evolution and model migration steps. Custom operations can be written and re-used.

[5] `http://groovy.codehaus.org/`

Ecore2Ecore. Mapping models cannot be reused or extended in Ecore2Ecore but as the custom model parser is specified in Java, developers can decompose it into reusable parts some of which can potentially be reused in other migrations.

Flock. A migration strategy encoded in Flock is modularised according to the classes whose instances need migration. There is support to reuse code within a strategy by means of operations with parameters and across strategies by means of imports. Re-use in Flock captures only migration patterns, and not the higher level co-evolution patterns captured in COPE or AML.

Conciseness

A concise migration strategy is arguably more readable and requires less effort to write than a verbose migration strategy. This section comments on the conciseness of migration strategies produced with each tool, and reports the lines of code (without comments and blank lines) used.

AML. 117 lines were automatically generated for the Petri nets example. 563 lines were automatically generated for the GMF Graph example, and a further 63 lines of code were added by hand to complete the transformation. Approximately 10 lines of the user-defined code could be removed by restructuring the generated transformation.

COPE requires the user to apply operations. Each operation application generates one line of code. The user may also write additional migration code. For the Petri net example, 11 operations were required to create the migrator and no additional code. The author of COPE migrated the GMF Graph example using 76 operations and 73 lines of additional code.

Ecore2Ecore. As discussed above, handling changes that cannot be declared in the mapping model is a tedious task and involves a significant amount of low level code. For the PetriNets example, the Ecore2Ecore solution involved a mapping model containing 57 lines of (automatically generated) XMI and a custom hand-written resource handler containing 78 lines of Java code.

Flock. 16 lines of code were necessary to encode the Petri nets example, and 140 lines of code were necessary to encode the GMF Graph example. In the GMF Graph example, approximately 60 lines of code implement missing built-in support for rule inheritance, even after duplication was removed by extracting and re-using a subroutine.

Clarity

Because migration strategies can change and might serve as documentation for the history of a metamodel, their clarity is important. This section reports on aspects of each tool that might affect the clarity of migration strategies.

AML. The AML code generator takes a conservative approach to naming variables, to minimise the chances of duplicate variable names. Hence, some of the generated code can be difficult to read and hard to re-use if the generated transformation has to be completed by hand. When a complete transformation can be generated by AML, clarity is not as important.

COPE. Migration strategies in COPE are defined as a sequence of operations. The release history stores the set of operations that have been applied, so the user is clearly able to see the changes they have made, and find where any issues may have been introduced.

Ecore2Ecore. The graphical mapping editor provided by Ecore2Ecore allows developers to have a high-level visual overview of the simple mappings involved in the migration. However, migrations expressed in the Java part of the solution can be far more obscure and difficult to understand as they mix high-level intention with low-level string management operations.

Flock clearly states the migration strategy from the source to the target metamodel. However, the boilerplate code necessary to implement rule inheritance slightly obfuscates the real migration code.

Expressiveness

Migration strategies are easier to infer for some categories of metamodel change than others [Gruschko *et al.* 2007]. This section reports on the ability of each tool to migrate the examples considered in this comparison.

AML. A complete migrating transformation could be generated for the Petri nets example, but not for the GMF Graph example. The latter contains examples of two complex changes that AML does not currently support[6]. Successfully expressing the GMF Graph example in AML would require changes to at least one of AML's heuristics. However, AML provided an initial migration transformation that was completed by hand.

In general, AML cannot be used to generate complete migration strategies for co-evolution examples that contain *breaking and non-resolvable changes*, according to the categorisation proposed in [Gruschko *et al.* 2007].

[6] `http://www.eclipse.org/forums/index.php?t=rview&goto=526894#msg_526894If`

COPE. The expressiveness of COPE is defined by the set of operations available. The Petri net example was migrated using only built-in operations. The GMF Graph example was migrated using 76 built-in operations and 2 user-defined migration actions. Custom migration actions allow users to specify any migration strategy.

Ecore2Ecore. A complete migration strategy could be generated for the Petri nets example, but not for the GMF Graph example. The developers of Ecore2Ecore have advised that the latter involves significant structural changes between the two versions and recommended implementing a custom model parser from scratch.

Flock. Since Flock extends EOL, it is expressive enough to encode both examples. However, Flock does not provide an explicit construct to copy model elements and thus it was necessary to call Java code from within Flock for the GMF Graph example.

Interoperability

Migration occurs in a variety of settings with differing requirements. This section considers the technical dependencies and procedural assumptions of each tool, and seeks to answer questions such as: "Which modelling technologies can be used?" and "What assumptions does the tool make on the migration process?"

AML depends only on ATL, while its development tools also require Eclipse. AML assumes that the original and target metamodels are available for comparison, and does not require a record of metamodel changes. AML can be used with either Ecore (EMF) or KM3 metamodels.

COPE depends on EMF and Groovy, while its development tools also require Eclipse and EMF Compare. COPE does not require both the original and target metamodels to be available. When COPE is used to create a migration strategy after metamodel evolution has already occurred, the metamodel changes must be reverse-engineered. To facilitate this, the target metamodel can be used with the Convergence View, as discussed in Section 6.3.2. COPE targets EMF, and does not support other modelling technologies.

Ecore2Ecore depends only on EMF. Both the original and the evolved versions of the metamodel are required to specify the mapping model with the Ecore2Ecore development tools. Alternatively, the Ecore2Ecore mapping model can be constructed programmatically and without using the original

metamodel[7]. Unlike the other tools considered, Ecore2Ecore does not require the original metamodel to be available in the workspace of the metamodel user.

Flock depends on Epsilon and its development tools also require Eclipse. Flock assumes that the original and target metamodels are available for encoding the migration strategy, and does not require a record of metamodel changes. Flock can be be used to migrate models represented in EMF, MDR, XML and Z (CZT), although we only encoded a migration strategy for EMF metamodels in the presented examples.

Performance

The time taken to execute model migration is important, particularly once a migration strategy has been distributed to metamodel users. Ideally, migration tools will produce migration strategies whose execution time is quick and scales well with large models.

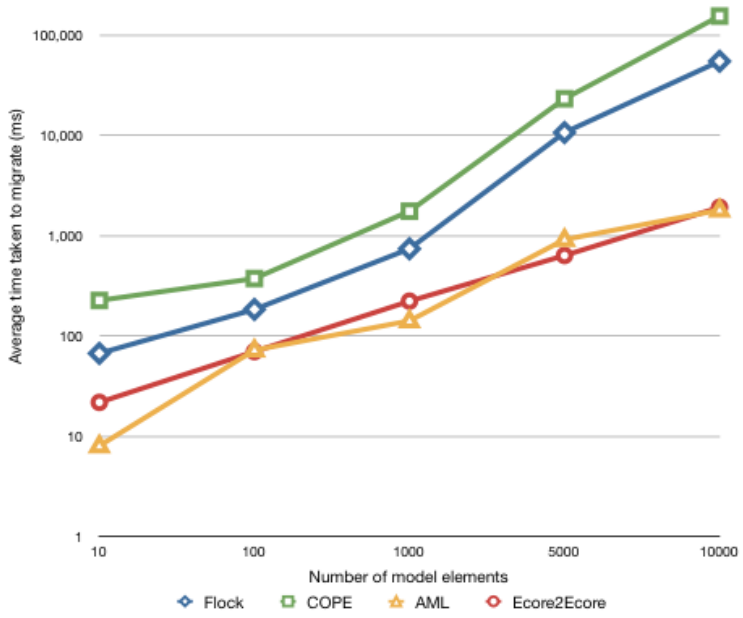


Figure 6.15: Migration tool performance comparison.

To measure performance, five sets of Petri net models were generated at random. Models in each set contained 10, 100, 1000, 5,000, and 10,000 model elements. Figure 6.15 shows the average time taken by each tool to execute migration across 10 repetitions for models of different sizes. Note that the

[7]Private communication with Marcelo Paternostro, an Ecore2Ecore developers.

Y axis has a logarithmic scale. The results indicate that, for the Petri nets co-evolution example, AML and Ecore2Ecore execute migration significantly more quickly than COPE and Flock, particularly when the model to be migrated contains more than 1,000 model elements. Figure 6.15 indicates that, for the Petri nets co-evolution example, Flock executes migration between two and three times faster than COPE, although the author of COPE reports that turning off validation causes COPE to perform similarly to Flock.

6.3.3 Discussion

The comparison described above highlights similarities and differences between a representative sample of model migration approaches. From this comparison, guidance for selecting between tools was synthesised. The guidance is presented below, and was produced by all four participants in the comparison (Herrmannsdöerfer, Williams, Kolovos and myself).

COPE captures co-evolution patterns (which apply to both model and metamodel), while Ecore2Ecore, AML and Flock capture only model migration patterns (which apply just to models). Because of this, COPE facilitates a greater degree of re-use in model migration than other approaches. However, the order in which the user applies patterns with COPE impacts on both metamodel evolution and model migration, which can complicate pattern selection particularly when a large amount of evolution occurs at once. The re-usable co-evolution patterns in COPE make it well suited to migration problems in which metamodel evolution is frequent and in small steps.

Flock, AML and Ecore2Ecore are preferable to COPE when metamodel evolution has occurred before the selection of a migration approach. Because of its use of co-evolution patterns, we conclude that COPE is better suited to forward- rather than reverse-engineering.

Through its Convergence View and integration with the EMF metamodel editor, COPE facilitates metamodel analysis that is not possible with the other approaches considered in this paper. COPE is well-suited to situations in which measuring and reasoning about co-evolution is important.

In situations where migration involves modelling technologies other than EMF, AML and Flock are preferable to COPE and Ecore2Ecore. AML can be used with models represented in KM3, while Flock can be used with models represented in MDR, XML and CZT. Via the connectivity layer of Epsilon, Flock can be extended to support further modelling technologies.

There are situations in which Ecore2Ecore or AML might be preferable to Flock and COPE. For large models, Ecore2Ecore and AML might execute migration significantly more quickly than Flock and COPE. Ecore2Ecore is the only tool that has no technical dependencies (other than a modelling framework). In situations where migration must be embedded in another tool, Ecore2Ecore offers a smaller footprint than other migration approaches. Compared to the other approaches considered in this paper, AML automatically

generates migration strategies with the least guidance from the user.

Despite these advantages, Ecore2Ecore and AML are unsuitable for some types of migration problem, because they are less expressive than Flock and COPE. Specifically, changes to the containment of model elements typically cannot be expressed with Ecore2Ecore and changes that are classified by [Herrmannsdoerfer *et al.* 2008] as *metamodel-specific* cannot be expressed with AML. Because of this, it is important to investigate metamodel changes before selecting a migration tool. Furthermore, it might be necessary to anticipate which types of metamodel change are likely to arise before selecting a migration tool. Investing in one tool to discover later that it is no longer suitable causes wasted effort.

Table 6.4: Summary of tool selection advice. (Tools are ordered alphabetically).

Requirement	**Recommended Tools**
Frequent, incremental co-evolution	COPE
Reverse-engineering	AML, Ecore2Ecore, Flock
Modelling technology diversity	Flock
Quicker migration for larger models	AML, Ecore2Ecore
Minimal dependencies	Ecore2Ecore
Minimal hand-written code	AML, COPE
Minimal guidance from user	AML
Support for metamodel-specific migrations	COPE, Flock

6.3.4 Summary

The work presented in this section compared a representative sample of approaches to automating model migration. The comparison was performed by following a methodical process and using an example from a real-world MDE project. Some preliminary recommendations and guidelines in choosing a migration tool were synthesised from the presented results and are summarised in Table 6.4. The comparison was carried out by the developers of the migration tools (or stand-ins where the developers were unable to participate fully). Each developer used a tool other than their own so that the comparison could more closely emulate the level of expertise of a typical user.

Some criteria were excluded from the comparison because of the method employed. For instance, the learnability of a tool affects the productivity of users, and, as such, affects tool selection. However, drawing conclusions about learnability (and also productivity and usability) is challenging with the comparison method employed because of the subjective nature of these characteristics. A comprehensive user study (with hundreds of users) would be more suitable for assessing these types of criteria.

6.4 Transformation Tools Contest

The Transformation Tools Contest (TTC) is a workshop series that seeks to compare and contrast tools for performing model and graph transformation. At TTC 2010, two rounds of submissions were invited: cases (transformation problems, three of which are selected by the workshop organisers) and solutions to the selected cases. In addition, TTC 2010 include a *live contest*: during the workshop a further transformation problem was announced and solutions submitted.

Participation in TTC 2010 facilitated further evaluation of Flock. Flock and 8 other transformation tools were assessed for a model migration problem based on a real-world example of metamodel evolution from the UML [OMG 2007b]. As part of the live contest, Flock was also assessed along with 13 transformation tools for a model transformation problem. Compared to the evaluation described in Section 6.3, the evaluation in this section compares Flock to a wider range of tools (model and graph transformation tools, and not just model migration tools), and investigates the suitability of Flock for specifying model transformation (and not just model migration).

The remainder of this section describes the model migration problem (Section 6.4.1) and Flock solution (Section 6.4.2), and the use of Flock for specifying a model transformation in the live contest (Section 6.4.3).

6.4.1 Model Migration Case

To compare Flock with other transformation tools for specifying model migration, the thesis author submitted a case to TTC based on the evolution of the UML. The way in which activity diagrams are modelled in the UML changed significantly between versions 1.4 and 2.1 of the specification. In the former, activities were defined as a special case of state machines, while in the latter they are defined atop a more general semantic base[8] [Selic 2005].

The remainder of this section briefly introduces UML activity diagrams, describes their evolution, and discusses the way in which solutions were assessed. The work presented in this section is based on the case submitted to TTC 2010 [Rose *et al.* 2010d].

Activity Diagrams in UML

Activity diagrams are used for modelling lower-level behaviours, emphasising sequencing and co-ordination conditions. They are used to model business processes and logic [OMG 2007b]. Figure 6.16 shows an activity diagram for filling orders. The diagrams is partitioned into three *swimlanes*, representing different organisational units. *Activities* are represented with rounded rectangles and *transitions* with directed arrows. *Fork* and *join* nodes are specified

[8] A variant of generalised coloured Petri nets.

using a solid black rectangle. *Decision* nodes are represented with a diamond. Guards on transitions are specified using square brackets. For example, in Figure 6.16 the transition to the restock activity is guarded by the condition `[not in stock]`. Text on transitions that is not enclosed in square brackets represents a trigger event. In Figure 6.16, the transition from the restock activity occurs on receipt of the asynchronous signal called `receive stock`. Finally, the transitions between activities might involve interaction with objects. In Figure 6.16, the Fill Order activity leads to an interaction with an object called `Filled Object`.

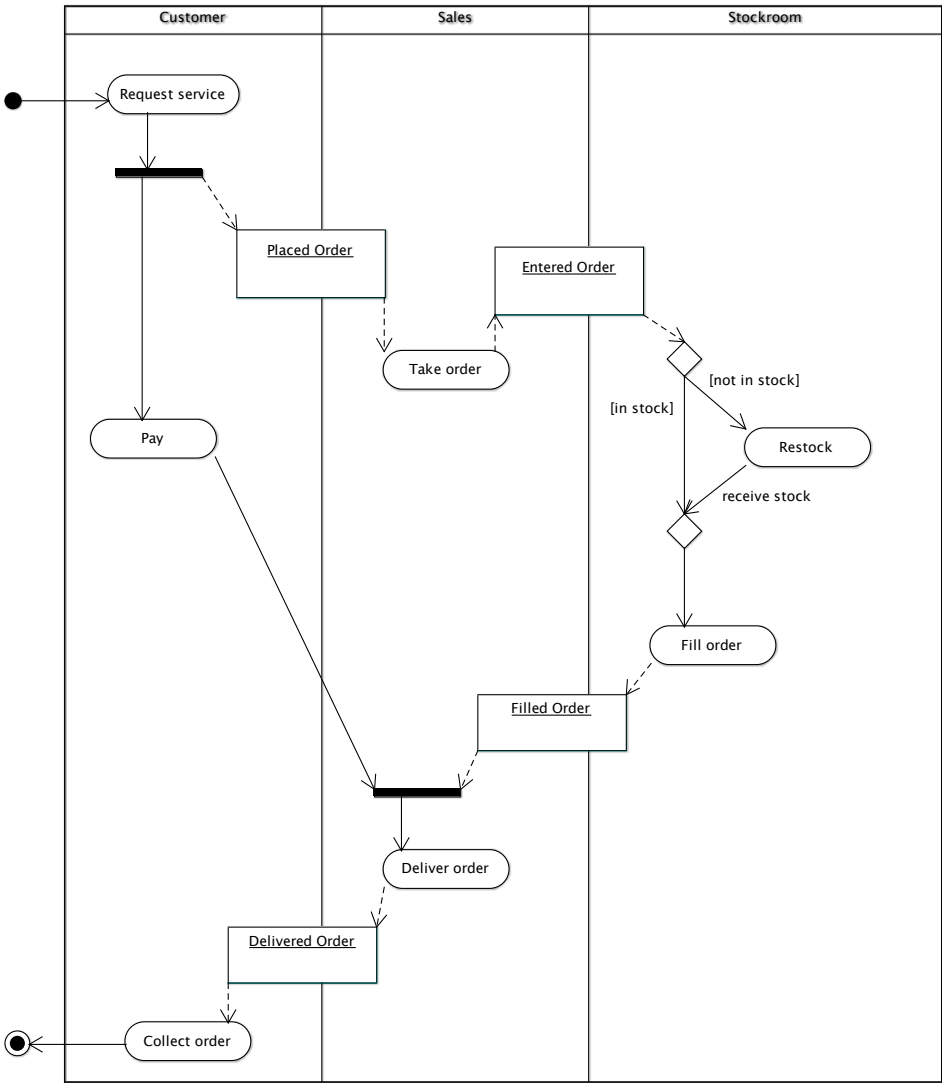


Figure 6.16: Exemplar activity model.

Between versions 1.4 and 2.2 of the UML specification, the metamodel for activity diagrams has changed significantly. The sequel summarises most of the changes, and details can be found in [OMG 2001] and [OMG 2007b].

Evolution of Activity Diagrams

Figures 6.17 and 6.18 are simplifications of the activity diagram metamodels from versions 1.4 and 2.2 of the UML specification, respectively. In the interest of clarity, some features and abstract classes have been removed from Figures 6.17 and 6.18.

Some differences between Figures 6.17 and 6.18 are: activities have been changed such that they comprise nodes and edges, actions replace states in UML 2.2, and the subtypes of control node replace pseudostates.

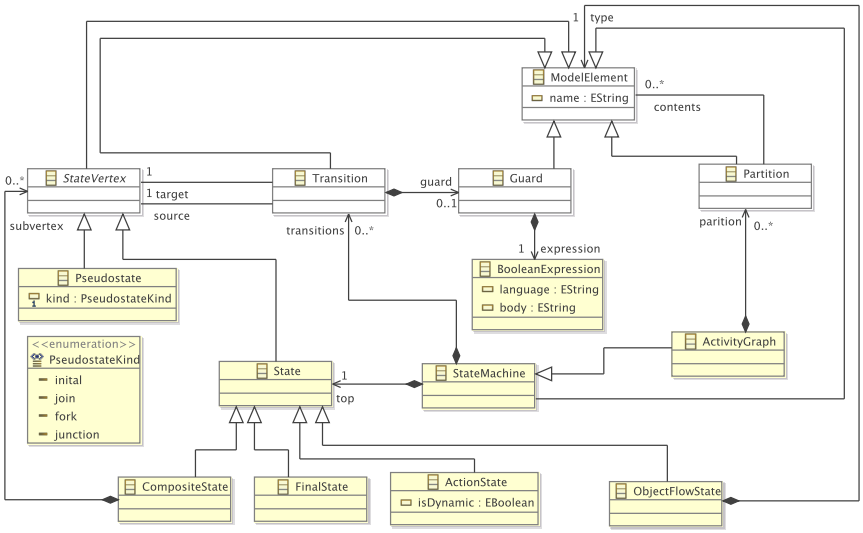


Figure 6.17: UML 1.4 Activity Graphs (based on [OMG 2001]).

To facilitate the comparison of solutions, the exemplar model shown in Figure 6.16 was used. Figure 6.16 is based on [OMG 2001, pg3-165]. Solutions migrated the activity diagram shown in Figure 6.16 – which conforms to UML 1.4 – to conform to UML 2.2. The UML 1.4 model, the migrated UML 2.2 model, and the UML 1.4 and 2.2 metamodels are available from[9].

Submissions were evaluated using the following four criteria, which were decided by the thesis author and the workshop organisers:

- **Correctness**: Does the transformation produce a model equivalent to the migrated UML 2.2. model included in the case resources?

[9] `http://www.cs.york.ac.uk/~louis/ttc/`

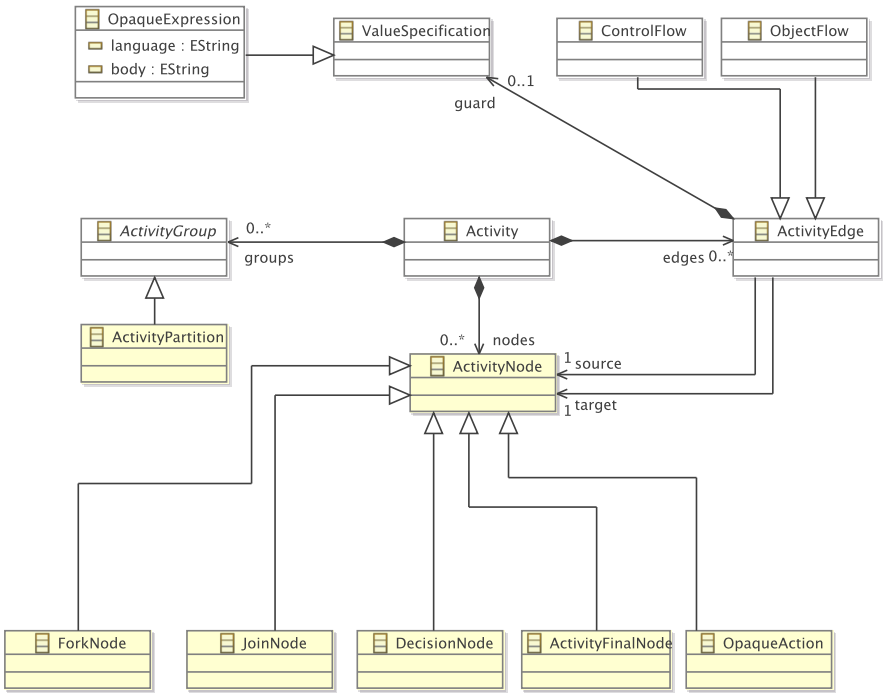


Figure 6.18: UML 2.2 Activity Diagrams (based on [OMG 2007b]).

- **Conciseness**: How much code is required to specify the transformation? (In [Sprinkle & Karsai 2004] et al. propose that the amount of effort required to codify migration should be directly proportional to the number of changes between original and evolved metamodel).

- **Clarity**: How easy is it to read and understand the transformation? (For example, is a well-known or standardised language?)

- **Extensions**: Which of the case extensions (described below) were implemented in the solution?

To further distinguish between solutions, three extensions to the core task were proposed. The first extension was added after the case was submitted, and was proposed by the workshop organisers and the solution authors. The second and third extension were included in the case by the thesis author.

Extension 1: Alternative Object Flow State Migration Semantics

Following the submission of the case, discussion on the TTC forums[10] revealed an ambiguity in the UML 2.2 specification indicating that the migration semantics for the ObjectFlowState UML 1.4 concept are not clear from the UML 2.2 specification.

In the core task described above, instances of ObjectFlowState were migrated to instances of ObjectNode. Any instances of Transition that had an ObjectFlowState as their source or target were migrated to instances of ObjectFlow. Listing 6.7 shows an example application of this migration semantics. The top line of Listing 6.7 shows instances of UML 1.4 metaclasses, include an instance of ObjectFlowState. The bottom line of Listing 6.7 shows the equivalent UML 2.2 instances according to this migration semantics. Note that the Transitions, t1 and t2, are migrated to an instance of ObjectFlow. Likewise, the instance of ObjectFlowState, s2, is migrated to an instance of ObjectFlow.

```
1 s1:State <- t1:Transition -> s2:ObjectFlowState <- t2:Transition -> s3:
      State
2
3 s1:ActivityNode <- t1:ObjectFlow -> s2:ObjectNode <- t2:ObjectFlow ->
      s3:ActivityNode
```

Listing 6.7: Migrating Actions

This extension considered an alternative migration semantics for ObjectFlowState. For this extension, instances of ObjectFlowState (and any connected Transitions) were migrated to instances ObjectFlow, as shown by the example in Listing 6.8 in which the UML 2.2 ObjectFlow, f1, replaces t1, t2 and s2.

```
1 s1:State <- t1:Transition -> s2:ObjectFlowState <- t2:Transition -> s3:
      State
2
3 s1:ActivityNode <- f1:ObjectFlow -> s3:ActivityNode
```

Listing 6.8: Migrating Actions

The alternative semantics were proposed on the TTC 2010 forums, and agreed as an extension to the core task by consensus between the solution authors and the workshop organisers.

Extension 2: Concrete Syntax

The second extension relates to the appearance of activity diagrams. The UML specifications provide no formally defined metamodel for the concrete

[10] http://planet-research20.org/ttc2010/index.php?option=com_community&view=groups&task=viewgroup&groupid=4&Itemid=150 (registration required)

syntax of UML diagrams. However, some UML tools store diagrammatic information in a structured manner using XML or a modelling tool. For example, the Eclipse UML 2 tools [Eclipse 2009b] store diagrams as GMF [Gronback 2009] diagram models.

As such, submissions were invited to explore the feasibility of migrating the concrete syntax of the activity diagram shown in Figure 6.16 to the concrete syntax in their chosen UML 2 tool. To facilitate this, the case resources included an ArgoUML project[11] containing the activity diagram shown in Figure 6.16.

Extension 3: XMI

The UML specifications indicate that UML models should be stored using XMI. However, because XMI has evolved at the same time as UML, UML 1.4 tools most likely produce XMI of a different version to UML 2.2 tools. For instance, ArgoUML produces XMI 1.2 for UML 1.4 models, while the Eclipse UML2 tools produce XMI 2.1 for UML 2.2.

As an extension to the core task, submissions were invited to consider how to migrate a UML 1.4 model represented in XMI 1.x to a UML 2.1 model represented in XMI 2.x. To facilitate this, the UML 1.4 model shown in Figure 6.16 was made available in XMI 1.2 as part of the case resources.

Following the submission of the case, solutions were encouraged to solve this extension by Tom Morris, the project leader for ArgoEclipse and a committer on ArgoUML. On the TTC forums, Morris stated that "We have nothing available to fill this hole currently, so any contributions would be hugely valuable. Not only would achieve academic fame and glory from the contest, but you'd get to see your code benefit users of one of the oldest (10+ yrs) open source UML modeling tools." [12]

6.4.2 Model Migration Solution in Epsilon Flock

This section discusses the Flock solution to the TTC case described above (the evolution of UML activity diagrams).

The solution was developed in an iterative and incremental manner, using the following process:

1. Change the Flock migration strategy.
2. Execute Flock on the original model, producing a migrated model.
3. Compare the migrated model with the reference model provided in the case resources.

[11] `http://argouml.tigris.org/`

[12] `http://www.planet-research20.org/ttc2010/index.php?option=com_community&view=groups&task=viewdiscussion&groupid=4&topicid=20&Itemid=150` (registration required)

4. Repeat until the migrated and reference models were the same.

The remainder of this section presents the Flock solution in an incremental manner. The code listings in this section show only those rules relevant to the iteration being discussed.

Actions, Transitions and Final States

Development of the migration strategy began by executing an empty Flock migration strategy on the original model. Because Flock automatically copies model elements that have not been affected by evolution, the resulting model contained `Pseudostatess` and `Transitions`, but none of the `ActionStates` from the original model. In UML 2.2 activities, `OpaqueActions` replace `ActionStates`. Listing 6.9 shows the Flock code for changing `ActionStates` to corresponding `OpaqueActions`.

```
1 migrate ActionState to OpaqueAction
```

Listing 6.9: Migrating Actions

Next, similar rules were added to migrate instances of `FinalState` to instances of `ActivityFinalNode` and to migrate instances of `Transition` to `ControlFlow`, as shown in Listing 6.10.

```
1 migrate FinalState to ActivityFinalNode
2 migrate Transition to ControlFlow
```

Listing 6.10: Migrating FinalStates and Transitions

Pseudostates

Development continued by selected further types of state that were not present in the migrated model, such as `Pseudostatess`, which are not used in UML 2.2 activities. Instead, UML 2.2 activities use specialised `Nodes`, such as `InitialNode`. Listing 6.11 shows the Flock code used to change `Pseudostates` to corresponding `Nodes`.

```
1 migrate Pseudostate to InitialNode when: original.kind = Original!
      PseudostateKind#initial
2 migrate Pseudostate to DecisionNode when: original.kind = Original!
      PseudostateKind#junction
3 migrate Pseudostate to ForkNode when: original.kind = Original!
      PseudostateKind#fork
4 migrate Pseudostate to JoinNode when: original.kind = Original!
      PseudostateKind#join
```

Listing 6.11: Migrating Pseudostates

Activities

In UML 2.2, `Activitys` no longer inherit from state machines. As such, some of the features defined by `Activity` have been renamed. Specifically, `transitions` has become `edges` and `paritions` has become `group`. Furthermore, the states (or nodes in UML 2.2 parlance) of an `Activity` are now contained in a feature called `nodes`, rather than in the `subvertex` feature of a composite state accessed via the `top` feature of `Activity`. The Flock migration rule shown in Listing 6.12 captured these changes.

```
1 migrate ActivityGraph to Activity {
2   migrated.edge = original.transitions.equivalent();
3   migrated.group = original.partition.equivalent();
4   migrated.node = original.top.subvertex.equivalent();
5 }
```

Listing 6.12: Migrating ActivityGraphs

Note that the rule in Listing 6.12 used the built-in `equivalent` operation to find migrated model elements from original model elements. As discussed in Section 5.4, the `equivalent` operation invokes other migration rules where necessary and caches results to improve performance.

Next, a similar rule for migrating `Guards` was added. In UML 1.4, the the `guard` feature of `Transition` references a `Guard`, which in turn references an `Expression` via its `expression` feature. In UML 2.2, the `guard` feature of `Transition` references an `OpaqueExpression` directly. Listing 6.13 captures this in Flock.

```
1 migrate Guard to OpaqueExpression {
2   migrated.body.add(original.expression.body);
3 }
```

Listing 6.13: Migrating Guards

Partitions

In UML 1.4 activity diagrams, `Partition` specifies a single containment reference for its `contents`. In UML 2.2 activity diagrams, partitions have been renamed to `ActivityPartitions` and specify two containment features for their contents, `edges` and `nodes`. Listing 6.14 shows the rule used to migrate `Partitions` to `ActivityPartitions` in Flock. The body of the rule shown in Listing 6.14 uses the *collect* operation to segregate the `contents` feature of the original model element into two parts.

```
1 migrate Partition to ActivityPartition {
2   migrated.edges = original.contents.collect(e:Transition | e.equivalent
        ());
```

```
3   migrated.nodes = original.contents.collect(n:StateVertex | n.
        equivalent());
4 }
```

Listing 6.14: Migrating Partitions

ObjectFlows

Finally, two rules were written for migrating model elements relating to object flows. In UML 1.4 activity diagrams, object flows are specified using `ObjectFlowState`, a subtype of `StateVertex`. In UML 2.2 activity diagrams, object flows are modelled using a subtype of `ObjectNode`. In UML 2.2 flows that connect to and from `ObjectNodes` must be represented with `ObjectFlows` rather than `ControlFlows`.

Listing 6.15 shows the Flock rule used to migrate `Transitons` to `ObjectFlows`. The rule applies for `Transitions` whose source or target `StateVertex` is of type `ObjectFlowState`.

```
1 migrate ObjectFlowState to ActivityParameterNode
2
3 migrate Transition to ObjectFlow when: original.source.isTypeOf(
      ObjectFlowState) or original.target.isTypeOf(ObjectFlowState)
```

Listing 6.15: Migrating ObjectFlows

In addition to the core task, the Flock solution also approached two of the three extensions described in the case (Section 6.4.1). The solutions to the extensions are now discussed.

Alternative ObjectFlowState Migration Semantics

The first extension required submissions to consider an alternative migration semantics for `ObjectFlowState`, in which a single `ObjectFlow` replaces each `ObjectFlowState` and any connected `Transitions`.

Listing 6.16 shows the Flock source code used to migrate `ObjectFlowStates` (and connecting `Transitions`) to a single `ObjectFlow`. This rule was used instead of the two rules defined in Listing 6.15. In the body of the rule shown in Listing 6.16, the `source` of the `Transition` is copied directly to the `source` of the `ObjectFlow`. The `target` of the `ObjectFlow` is set to the target of the first outgoing `Transition` from the `ObjectFlowState`.

```
1 migrate Transition to ObjectFlow when: original.target.isTypeOf(
      ObjectFlowState) {
2   migrated.source = original.source.equivalent();
3   migrated.target = original.target.outgoing.first.target.equivalent();
4 }
```

Listing 6.16: Migrating ObjectFlowStates to a single ObjectFlow

Because, in this alternative semantics, `ObjectFlowStates` are represented as edges rather than nodes, the partition migration rule was changed such that `ObjectFlowStates` were not copied to the `nodes` feature of `Partitions`. To filter out the `ObjectFlowStates`, line 3 of Listing 6.14 was changed to include a reject statement, as shown on line 3 of Listing 6.17.

```
1 migrate Partition to ActivityPartition {
2   migrated.edges = original.contents.collect(e:Transition | e.equivalent
        ());
3   migrated.nodes = original.contents.reject(ofs:ObjectFlowState | true).
        collect(n:Original!StateVertex | n.equivalent());
4 }
```

Listing 6.17: Migrating Partitions without ObjectFlowStates

XMI

The second extension required submissions to migrate an activity graph conforming to UML 1.4 and encoded in XMI 1.2 to an equivalent activity graph conforming to UML 2.2 and encoded in XMI 2.1. The core task did not require submissions to consider changes to XMI (the model storage representation), but, in practice, this is a challenge to migration, as noted by Tom Morris on the TTC forums[13].

As discussed in Section 5.4, Flock is built atop Epsilon, which includes a model connectivity layer (EMC). EMC provides a common interface for accessing and persisting models. Currently, EMC supports EMF (XMI 2.x), MDR (XMI 1.x), and plain XML models. To support migration between metamodels defined in heterogenous modelling frameworks, EMC was extended during the development of Flock to provide a conformance checking service.

Consequently, the migration strategy developed for the core task works for all of the types of model supported by EMC. To migrate a model encoded in XMI 1.2 rather than in XMI 2.1, the user must select a different option when executing the Flock migration strategy. Otherwise, no other changes are required.

Results

At the workshop, solutions to the migration case described in Section 6.4.1 were presented. Each solution was allocated two opponents who highlighted weaknesses of each approach. Following the solution presentations and opposition statements, each solution was scored using the four criteria described

[13] `http://www.planet-research20.org/ttc2010/index.php?option=com_community&view=groups&task=viewdiscussion&groupid=4&topicid=20&Itemid=150` (registration required)

above, correctness, clarity, conciseness and number of extensions solved. Every workshop participants scored each solution on clarity and conciseness. The workshop organisers scored each solution on correctness and number of extensions solved, as these criteria could be measured objectively. Epsilon Flock was awarded first position for the migration case.

The opposition statements highlighted two weaknesses of Flock. Firstly, there is some duplicated code in Listing 6.11: the `migrate Pseudostate to X` statement appears several times. The duplication exists because Flock only allows one-to-one mappings between original and evolved metamodel types. The conservative copy algorithm would need to be extended to allow one-to-many mappings to remove this kind of duplication.

Secondly, the body of Flock rules are specified in an imperative manner. Consequently, reasoning about the correctness of the a migration strategy is arguably more difficult than in languages that use a purely declarative syntax. This point is discussed further in Section 6.5, which considers the limitations of the thesis.

6.4.3 Epsilon Flock in the Live Contest

TTC 2010 also invited the workshop participants to take part in a live contest. A problem was announced at the start of the workshop, and participants developed their solutions during the first day. The solutions were presented in the workshop and assessed in four categories. Flock was awarded first position for the *exogenous transformation* category. The remainder of this section discusses the parts of the problem that relate to the exogenous transformation category and the Flock solution.

The live contest problem required several model management operations be combined to perform beta-reduction of a simplified lambda calculus. Flock was used to specify one of the model management operations: model transformation between two similar (but not identical) metamodels. Flock was chosen rather than a new-target transformation language because the metamodels shared several classes and features. Using Flock allowed automatic copying of the model elements conforming to the classes common to both source and target metamodel. In other words, transformation rules were specified only for those parts of the metamodels that differed.

Flock was awarded first position by the workshop participants and organisers for the category in which it was entered. Participation in the live contest highlighted that, in addition to model migration, Flock can be used for specifying model transformation. In particular, Flock was appropriate because the source and target metamodels were similar (having several classes and features in common) and the conservative copy strategy reduced the number of rules required to specify the transformation.

6.4.4 Summary

This section has discussed the way in which Flock was evaluated by participating in the 2010 edition of the Transformation Tools Contest (TTC). Flock was assessed by application to an example of migration from the UML and comparison with eight other model and graph transformation tools. Flock was awarded first prize by the workshop participants and organisers. Additionally, Flock was used as part of a solution to a live contest developed during the workshop. The live contest highlighted that Flock is suitable for specifying some types of model transformation (in particular, those in which the source and target metamodel have common classes and features), as Flock was awarded first prize in the exogenous transformation category.

In addition to evaluating Flock, the work described in this section provides three further contributions. Firstly, the migration case submitted to TTC 2010, described in Section 6.4.1 provides a real-world example of co-evolution for use in future comparisons of model migration tools. The case is based on the evolution of UML, between versions 1.4 and 2.2. The migration strategy was devised by analysis of the UML specification, and by discussion between workshop participants.

Secondly, the Flock solution to the migration case (Section 6.4.2) demonstrates the way in which a migration strategy can be constructed using Flock. In particular, Section 6.4.2 describes an iterative and incremental development process and indicates that an empty Flock migration strategy can provide a useful starting point for development.

Finally, Section 5.4 claims that Flock support several modelling technologies. The solution described in Section 6.4.2 demonstrates the way in which Flock can be used to migrate models over two modelling technologies: MDR (XMI 1.x) and EMF (XMI 2.x), and hence supports the claim made in Section 5.4.

6.5 Limitations

The limitations and threats to the validity of the thesis research are now discussed. Some of the shortcomings identified here are elaborated on in Section 7.2, which highlights areas of future work.

Generality The thesis research focuses on model-metamodel co-evolution, but, as discussed in Chapter 4, metamodel changes can affect artefacts other than models. Model management operations and model editors are specified using metamodel concepts and, consequently, are affected when a metamodel changes. The work presented in Chapter 5 focuses on migrating models in response to metamodel changes, and does not consider integration with tools for migrating model management operations and model editors. To reduce the effort required to manage the effects of metamodel changes, it seems reason-

able to envisage a unified approach that migrates models, model management operations, model editors, and other affected artefacts.

Reproducibility The analysis and evaluation presented in Chapters 4 and 6 respectively involved using migration tools to understand and assess their functionality. With the exceptions noted below, the work presented in these chapters is difficult to reproduce and therefore the results drawn are somewhat subjective. On the other hand, multiple approaches to analysis and evaluation have been taken, and the work has been published and subjected to peer review.

Not all of the work in Chapter 4 and 6 is difficult to reproduce. In particular, Section 4.2 describes limitations of existing migration tools and was derived from the experiments discussed in Appendix A. To aid reproducibility, evaluation methods are described in detail in Sections 6.2 and 6.3. In general, the lack of real-world examples of co-evolution restricts the extent to which any work in this area can be considered reproducible.

Formal semantics No formal semantics for the conservative copy algorithm (Section 5.4) have been provided. Instead, a reference implementation, Epsilon Flock, was developed, which facilitated comparison with other migration and transformation tools. Without a reference implementation, the evaluation described in Sections 6.2, 6.3 and 6.4 would have been impossible. For Epsilon as a whole, [Kolovos 2009] makes a similar case for choosing a reference implementation over a formal semantics. For domains where completeness and correctness are a primary concern, a formal semantics would be required before Flock could be applied to manage model-metamodel co-evolution.

6.6 Summary

To be completed, but will include a paragraph similar to the following:

In addition to the evaluation described in this chapter, the work presented in this thesis has been subjected to peer review by the academic and Eclipse communities. The thesis research has been published in papers at XX workshops, YY European conferences and ZZ international conferences. HUTN, Flock and Concordance (Chapter 5) are part of the Epsilon project, a member of the research incubator for the Eclipse Modeling Project (EMP), which is arguably the most active MDE community at present. EMP's research incubator hosts a limited number of participants, selected through a rigorous process and contributions made to the incubator undergo regular technical review.

Chapter 7

Conclusion

7.1 Closing Remarks

- For software evolution research, need more data from industry. - Closer collaboration between industry and academia to study evolution in context. - More comprehensive assessment: for example, user studies.

7.2 Future Work

Generalisation

Relate to section in limitations. Two dimensions: – Seek a unified approach to migrating all artefacts that might be affected by metamodel changes (models, model management operations, model editors, etc). – Re-use concepts from model-metamodel co-evolution to manage other types of co-evolution. Summarise paper with Anne.

7.2.1 Extensions to Epsilon Flock

- Summarise limitations from Evaluation chapter.
- Discuss extensions to Flock to reduce duplication when subtyping.

  Try applying the changes suggested here[1] to the co-evo examples, and investigate whether they solve the subtyping problem / introduce other issues.
- Discuss extensions to conservative copy when a reference changes to a containment

[1] http://lmr109.basecamphq.com/projects/1508853/posts/30822433/comments

Appendix A

Experiments

- Introduction - Indicate that none of the literature makes analysis method available - Explain that not all examples were used in all experiments to save time

COPE v1.3.5 was used for these experiments.

A.1 Metamodel-Independent Change

Some metamodel changes are metamodel-independent and occur irrespective of the domain being modelled. Some examples of metamodel-independent changes are the renaming of a class, or changing a containment feature to a reference feature. The purpose of this experiment is to show the extent to which existing migration tools provide support for metamodel-independent changes, and to highlight any areas that could be improved. In particular, this experiment asks the following questions:

- Correctness: Do all migrated instance conform to the evolved metamodel?
- Automation: How much guidance was provided to the tool?
- Customisation: To what extent could the generated migration strategy be customised?
- Extensibility: How, if at all, could new metamodel-independent migration strategies be added?

The following steps for each evolutionary change were carried out in the co-evolution tool under analysis:

1. Recreate the metamodel as it was before any evolutionary change.
2. Adapt the metamodel according to the version history in its (real-world) project.

3. Use the tool to automatically migrate each instance of the original metamodel to conform to the evolved metamodel.

COPE was the only tool capable of performing migration using metamodel-independent changes. For this experiment, the object-oriented refactoring project was selected, as it contained some examples of co-evolution for which COPE provides a co-evolutionary operator, and some examples for which COPE does not.

A.1.1 Correctness

COPE had co-evolution operators for three of the seven examples: MoveFeature, Extract Class and Inline Class. Using the last two operators, COPE correctly migrated models in response to these refactorings. The co-evolution operation used for MoveFeature produced a conformant model, but with an unexpected side-effect: when two different values were specified for the same slot, the migration used the last value it encountered, rather than prompt the user or throw an error.

COPE provided no appropriate co-evolution operators for Change Containment to Reference, Change Reference to Containment, Extract Subclass and Push Down Feature. Since this experiment was conducted, I have contributed to COPE implementations for all but one of these operators. (It was not possible to implement an operator for Change Containment to Reference, as discussed below).

A.1.2 Automation

Very little data was supplied to use these refactorings. For example, for Extract Class, the name of the extracted class and the name of the reference was the only data required. Following metamodel evolution, COPE was used to generate a model migration (in the form of a Groovy script). The model migration depends on the COPE runtime, and, as such, COPE generates an Eclipse plugin for performing model migration. The plugin is distributed to metamodel users who wish to perform automated model migration.

A.1.3 Customisation

To customise a migration strategy, extra code could have been added to the generated Groovy script or Eclipse plug-in. However, the metamodel-independent change cannot be customised as it is implemented as a call to an operation embedded in the COPE runtime.

A.1.4 Extensibility

COPE provides an Eclipse extension point for contributing new metamodel-independent co-evolution operators. The operators must be authored in Groovy and be described in a corresponding instance of a metamodel provided by COPE. COPE provides some automation for generating instances of the description metamodel from Groovy code.

Authoring co-evolutionary operators for COPE required good knowledge of Ecore and some knowledge of Groovy. Groovy is weakly typed, and consequently type errors are reported at runtime. Testing co-evolutionary operators involves installing the extending plugin, specifying a metamodel evolution using the operators-under-test, generating a migration strategy plugin, installing the generated plugin and executing the generatjkkjed migration strategy on a model. This process is repeated each time an error is found, and COPE currently provides no way to test co-evolutionary operators in isolation.

To capture the co-evolution examples considered, a clone method for creating copies of model elements was implemented, which is not a feature provided by the COPE runtime. In addition, COPE provides no mechanisms for throwing an error or prompting the user for input, which would have been useful for the Move Feature example (as discussed above). It was not possible to implement a metamodel-independent migration strategy for Change Containment to Reference, as COPE does not support metamodel-independent migration strategies that require model elements as arguments.[1]

[1]TODO: Give an example or otherwise explain this point.

Appendix B

A Graphical Editor for Process-Oriented Programs

The way in which a prototypical graphical editor for process-oriented programs was designed and implemented is described in this appendix. The work presented here was conducted in collaboration with Adam Sampson, then a Research Associate at the University of Kent. The way in which the graphical editor changed throughout its development provided was used for evaluation of the thesis research in Section 6.1.

The purpose of the collaboration was to explore the suitability of MDE tools and techniques for designing a graphical notation and supporting tools for programs written in process-oriented programming languages, such as occam-π [Welch & Barnes 2005]. The collaboration resulted in the design and implementation of a prototypical graphical editor, which was used to construct examples of small process-oriented programs.

Process-oriented programs are specified in terms of three core concepts: processes, connection points and channels. Processes are the fundamental building blocks of a process-oriented program. Channels are the mechanism by which processes communicate, and are unidirectional. Connection points define the channels on which a process can communicate. Connection points are used to specify the way in which a process can communicate, and can optionally be bound to a channel. Because channels are unidirectional, connection points are either reading (consume messages from the channel) or writing (generate messages on the channel).

MDE tools and techniques were used to design and implement a graphical notation and editor for process-oriented programs by Sampson and the thesis author. An iterative style of development was used. The abstract syntax of the domain was specified as a metamodel, captured in Ecore, which is the metamodelling language of arguably the most widely-used contemporary MDE development environment, the Eclipse Modeling Framework (EMF) [Steinberg *et al.* 2008]. The graphical concrete syntax was specified with the

Graphical Modeling Framework (GMF) [Gronback 2006], via EuGENia, [Kolovos *et al.* 2009]. EMF and GMF are described more thoroughly in Section 2.3.

The remainder of this appendix presents the six versions of the process-oriented metamodel constructed by Sampson and the thesis author during the iterative development of the graphical editor. In addition, several models are also shown, which were used to test the graphical editor throughout its development.

B.1 Iteration 1: Processes and Channels

Development began by identifying two key concepts for modelling process-oriented programs. From examples of process-oriented programs, process and channel were identified as the most important concepts, and consequently the metamodel shown in Figure B.1 was constructed.

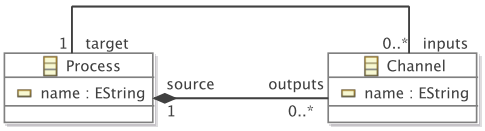


Figure B.1: The process-oriented metamodel after the first iteration.

Additionally, a graphical concrete syntax was chosen for processes and channels. The former were represented as boxes, and the latter as lines. EuGENia annotations were added to the metamodel, resulting in the metamodel shown in Listing B.1. Line 1 of Listing B.1 uses the "@gmf.node" EuGENia annotation to indicate that processes are to be represented as boxes with a label equal to the value of the `name` feature. Line 9 uses the "@gmf.link" EuGENia annotation to indicate that channels are to be represented as lines between `source` and `target` processes with a label equal to the value of the `name` feature.

```
1  @gmf.node(label="name")
2  class Process {
3    attr String name;
4
5    ref Channel[*]#target inputs;
6    val Channel[*]#source outputs;
7  }
8
9  @gmf.link(source="source", target="target", label="name")
10 class Channel {
11   attr String name;
12   ref Process[1]#outputs source;
```

```
13    ref Process[1]#inputs target;
14  }
```

Listing B.1: The annotated process-oriented metamodel after the first iteration

To generate code for the graphical editor, EuGENia was invoked on the annotated metamodel shown in Listing B.1. However, EuGENia failed with an error, because no "root" element had been specified. GMF, the graphical modelling framework used by EuGENia, requires one metaclass (termed the root) to be specified as a container for all diagram elements. The root metaclass cannot be a GMF node or a link, and so the second iteration involved adding an additional metaclass for interoperability with GMF.

B.2 Iteration 2: Interoperability with GMF

In the second iteration, an additional metaclass, `Model`, was added to the metamodel as shown in Figure B.2. The `Model` metaclass was used to provide GMF with a container for storing all of the diagram elements for each process-oriented diagram.

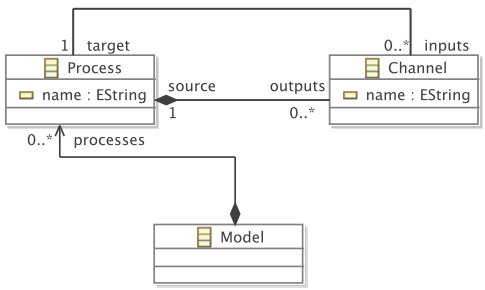


Figure B.2: The process-oriented metamodel after the second iteration.

As shown in Listing B.1, the `Model` metaclass was annotated with "@gmf.diagram" to indicate that it should be used as the diagram's root element. Root elements do not have a concrete syntax and do not appear in the graphical editor.

```
1  @gmf.diagram
2  class Model {
3    val Process[*] processes;
4  }
5
6  @gmf.node(label="name")
7  class Process {
8    attr String name;
```

```
9
10    ref Channel[*]#target inputs;
11    val Channel[*]#source outputs;
12  }
13
14
15  @gmf.link(source="source", target="target", label="name")
16  class Channel {
17    attr String name;
18    ref Process[1]#outputs source;
19    ref Process[1]#inputs target;
20  }
```

Listing B.2: The annotated process-oriented metamodel after the second iteration

EuGENia was invoked on the annotated metamodel shown in Listing B.2 to produce code for the graphical editor. Figure B.3 shows a simplistic test model, which was constructed to test the generated editor. The model shown in Figure B.3 comprises two processes, `P1` and `P2`, and one channel, `a`.

B.3 Iteration 3: Shared Channels

In previous iterations, channels had been contained within their source process. The nested structure made it more difficult to explore process-oriented models in EMF's tree editor due to the additional level of nesting. Consequently, the metamodel was changed such that channels were contained in the root element, rather than in the source process, resulting in the metamodel shown in Figure B.4.

No additional EuGENia annotations were added to the metamodel during this iteration. In other words, the graphical notation (concrete syntax) was not changed, and the resulting editor was identical in appearance to the previous one. However, the EMF tree editor showed just one level of nesting (everything is contained inside model).

The existing models required migration because of the way in which XMI differentiates between reference and containment values. Each channel was instantiated in the new `channels` reference of `Model`, and existing values in the `outputs` reference of `ConnectionPoint` were changed to use a reference rather than a containment value. Figure B.5(a) shows the HUTN for a model prior to migration, and Listing B.5(b) shows the reconciled, migrated HUTN.

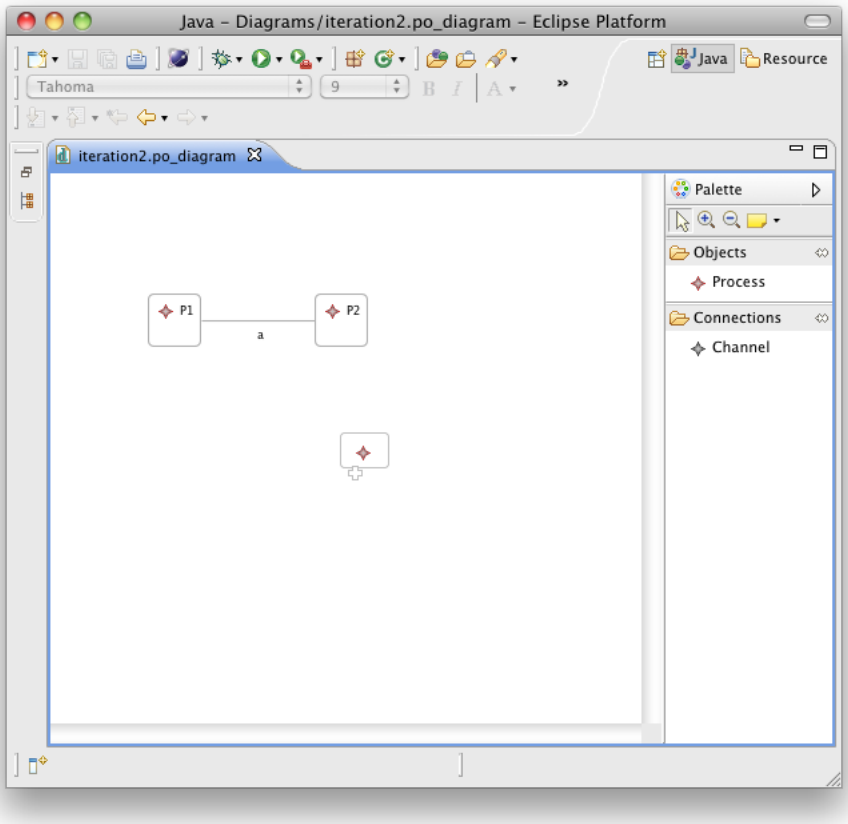


Figure B.3: Exemplar diagram after the second iteration.

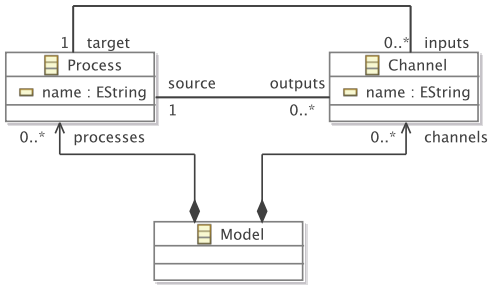
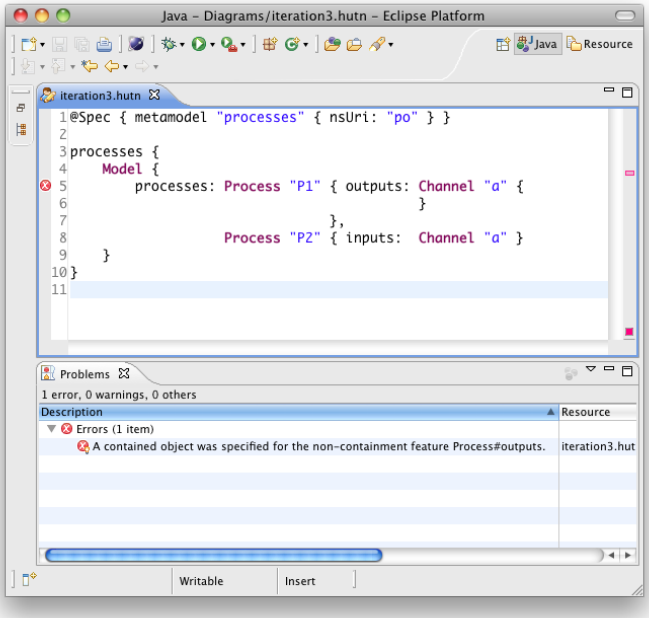
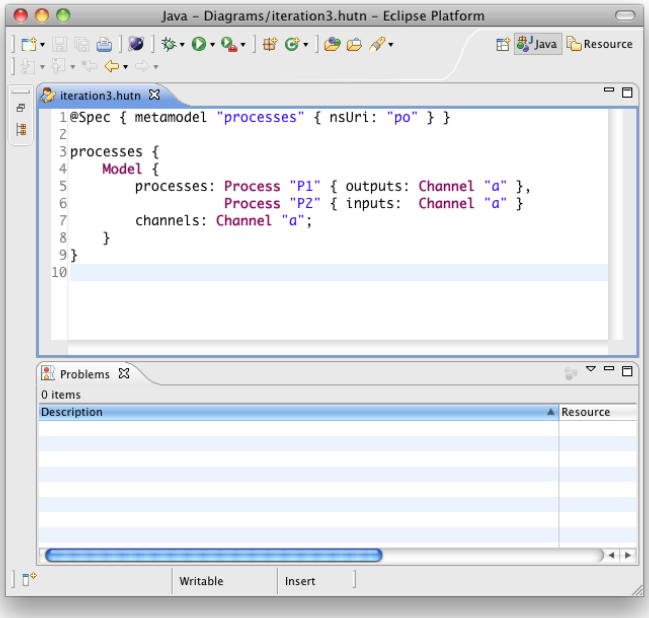


Figure B.4: The process-oriented metamodel after the third iteration.



(a) HUTN prior to migration



(b) HUTN after migration

Figure B.5: Exemplar migration between the second and third versions of the process-oriented metamodel

B.4 Iteration 4: Connection Points

The fourth iteration involved capturing a third domain concept, connection points, in the graphical notation. When a process is specified, the ways in which it can communicate are declared as connection points. When a process is instantiated, channels are connected to its connection points, and messages flow in and out of the process. The graphical notation was to be used to describe both instantiated processes and types of process, the metamodel was changed to model connection points.

The iteration resulted in the metamodel shown in Figure B.6. `ConnectionPoint` was introduced as an association class between the `inputs` and `outputs` references.

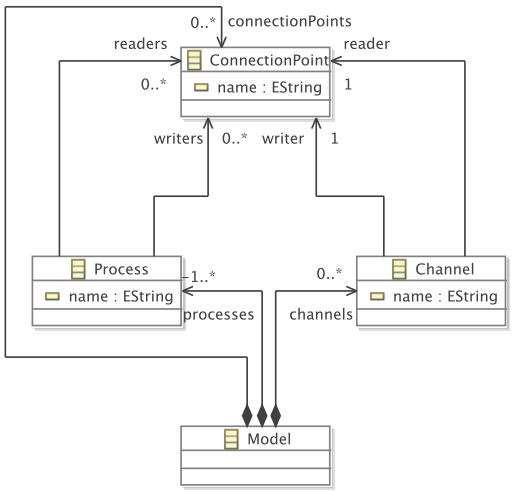


Figure B.6: The process-oriented metamodel after the fourth iteration.

To specify concrete syntax for connection points, additional EuGENia annotations were added to the metamodel as shown in Listing **??**. The `ConnectionPoint` class was annotated with a "@gmf.node" to specify that connections points were to be represented as circles, labelled with the value of the `name` attribute. The circles were to be affixed to the boxes used to represent processes, and, hence, "@gmf.affixed" annotations are used on lines 12 and 15.

```
1  @gmf.diagram
2  class Model {
3    val Process[*] processes;
4    val Channel[*] channels;
5    val ConnectionPoint[*] connectionPoints;
6  }
7
```

```
8  @gmf.node(label="name")
9  class Process {
10   attr String name;
11
12   @gmf.affixed
13   ref ConnectionPoint[*] readers;
14
15   @gmf.affixed
16   ref ConnectionPoint[*] writers;
17 }
18
19
20 @gmf.link(source="reader", target="writer", label="name", incoming="
       true")
21 class Channel {
22  attr String name;
23   ref ConnectionPoint[1] reader;
24   ref ConnectionPoint[1] writer;
25 }
26
27 @gmf.node(label="name", label.placement="external", label.icon="false",
        figure="ellipse", size="15,15")
28 class ConnectionPoint {
29   attr String name;
30 }
```

Listing B.3: The annotated process-oriented metamodel after the fourth iteration

A new version of the graphical editor was generated by invoking EuGENia on the annotated metamodel. A larger test model was constructed to test the editor, and is shown in Figure B.7. The existing models required migration because the `inputs` and `outputs` references of `Process` and the `source` and `target` references of `Channel` had been removed.

To migrate each existing model, two connection points were created for each channel in the model. The `source` and `target` reference of the channel was changed to reference the new connection points, as were the corresponding values of the `readers` and `writers` references of the relevant processes. Figure B.8(a) shows the HUTN for a model prior to migration, and Figure B.8(b) shows the reconciled, migrated HUTN.

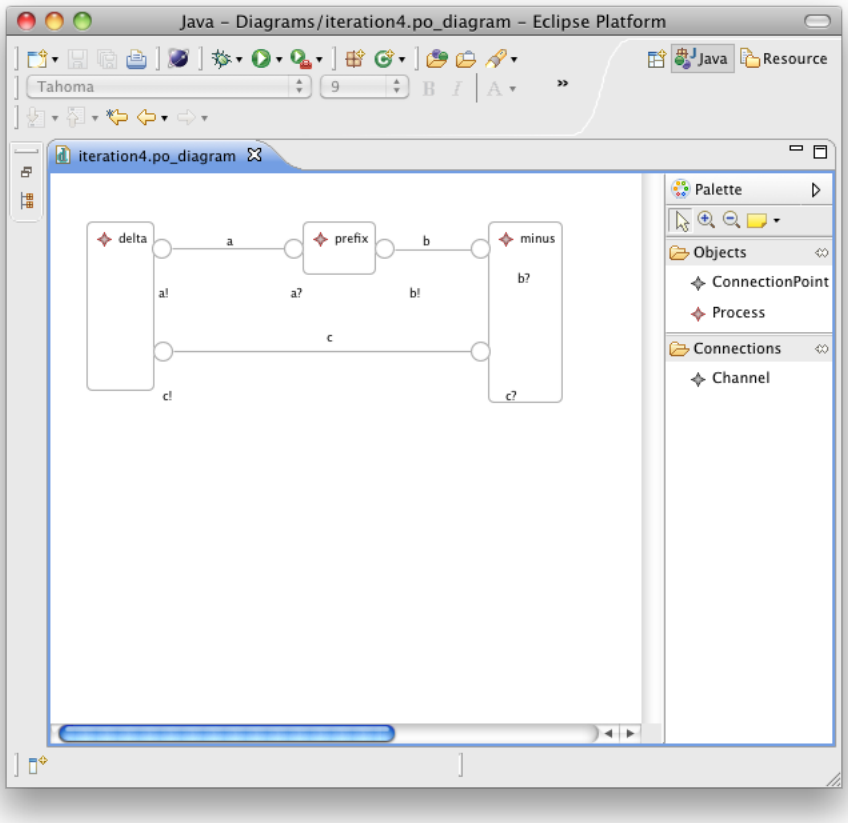
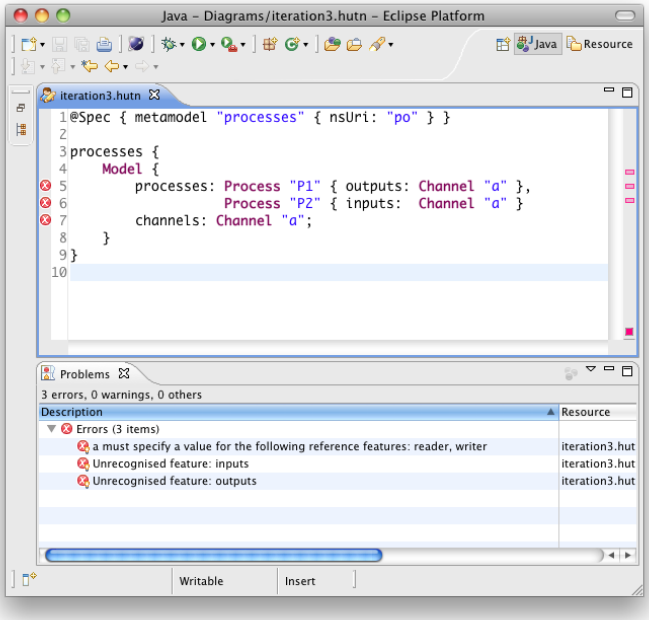
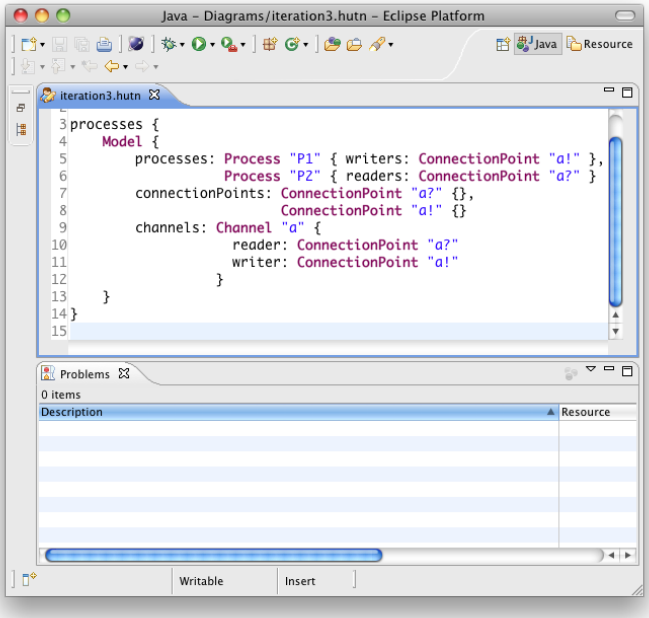


Figure B.7: Exemplar diagram after the fourth iteration.



(a) HUTN prior to migration



(b) HUTN after migration

Figure B.8: Exemplar migration between the third and fourth versions of the process-oriented metamodel

B.5 Iteration 5: Connection Point Types

Channels are unidirectional, and so connection points are either *reading* or *writing*. A process uses the former to consume messages from a channel, and the latter to produce messages on a channel. Testing the graphical editor producing in the fourth iteration showed that it was not immediately obvious as to which connection points were reading and which were writing. The fifth iteration involved changing the graphical editor to better distinguish between reading and writing connection points.

The iteration resulted in the metamodel shown in Figure B.9. `ConnectionPoint` was made abstract, and two subclass, `ReadingConnectionPoint` and `WritingConnectionPoint`, were introduced. The four references to `ConnectionPoint` were changed to reference one of the two subclasses.

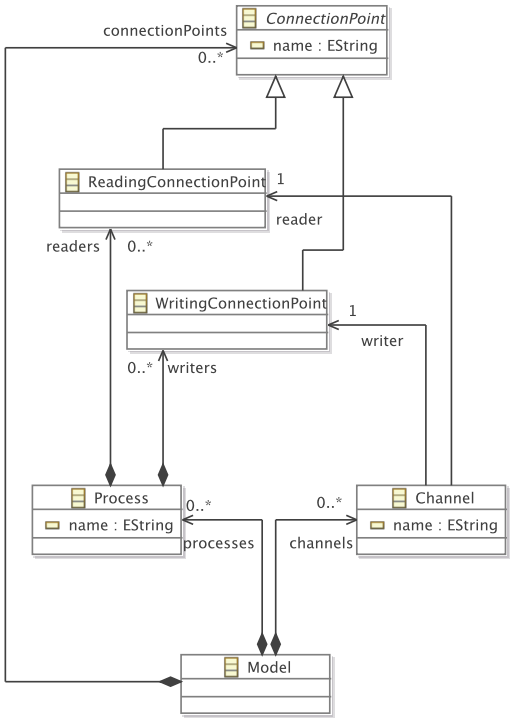


Figure B.9: The process-oriented metamodel after the fifth iteration.

The graphical notation was changed, as shown in Listing **??**. The `WritingConnectionPoint` class was annotated with an additional colour attribute to specify that writing connection points were to be represented with a black circle. White is the default colour for a "@gmf.node" annotation, and so reading connection points were represented as white circles.

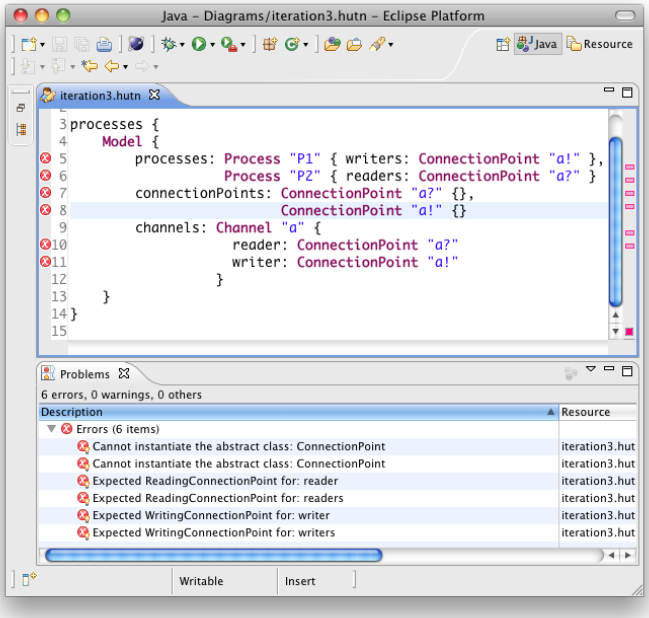
```
1  @gmf.diagram
```

```
2  class Model {
3    val Process[*] processes;
4    val Channel[*] channels;
5    val ConnectionPoint[*] connectionPoints;
6  }
7
8  @gmf.node(label="name")
9  class Process {
10   attr String name;
11
12   @gmf.affixed
13   ref ReadingConnectionPoint[*] readers;
14
15   @gmf.affixed
16   ref WritingConnectionPoint[*] writers;
17 }
18
19
20 @gmf.link(source="reader", target="writer", label="name", incoming="
       true")
21 class Channel {
22  attr String name;
23   ref ReadingConnectionPoint[1] reader;
24   ref WritingConnectionPoint[1] writer;
25 }
26
27 @gmf.node(label="name", label.placement="external", label.icon="false",
        figure="ellipse", size="15,15")
28 abstract class ConnectionPoint {
29   attr String name;
30 }
31
32 class ReadingConnectionPoint extends ConnectionPoint {}
33
34 @gmf.node(color="0,0,0")
35 class WritingConnectionPoint extends ConnectionPoint {}
```

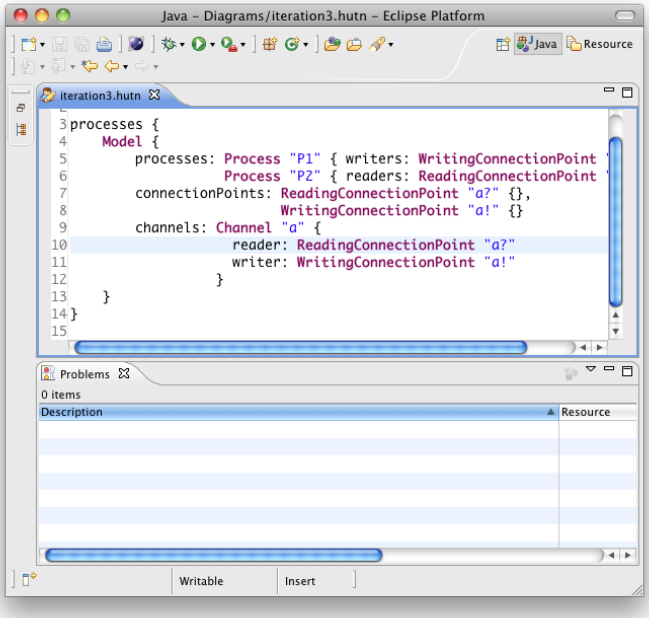
Listing B.4: The annotated process-oriented metamodel after the fifth iteration

A new version of the graphical editor was generated by invoking EuGENia on the annotated metamodel. All of the existing models required migration, because `ConnectionPoint` was now an abstract class, and could no longer be instantiated. Section 6.1 describes the way in which models were migrated after the changes made during this iteration. Briefly, migration involved replacing every instantiation of `ConnectionPoint` with an instantiation of either `ReadingConnectionPoint` or `WritingConnectionPoint`. The

former was used when a connection point was used as the value of a channel's `reader` feature and the latter when when a connection point was used as the value of a channel's `writer` feature. Listing B.10(a) shows the HUTN for a model prior to migration, and Listing B.10(b) shows the reconciled, migrated HUTN.



(a) HUTN prior to migration



(b) HUTN after migration

Figure B.10: Exemplar migration between the fourth and fifth versions of the process-oriented metamodel

B.6 Iteration 6: Nested Processes and Channels

The final iteration involved changing the graphical editor such that processes and channels could be nested inside other processes. In some process-oriented languages, such as occam-π [Welch & Barnes 2005], processes can be specified in terms of other, internal processes.

To support the decomposition of processes into other processes and channels, the `nestedProcess` and `nestedChannel` references were added to the `Process` class, as shown in Figure B.11.

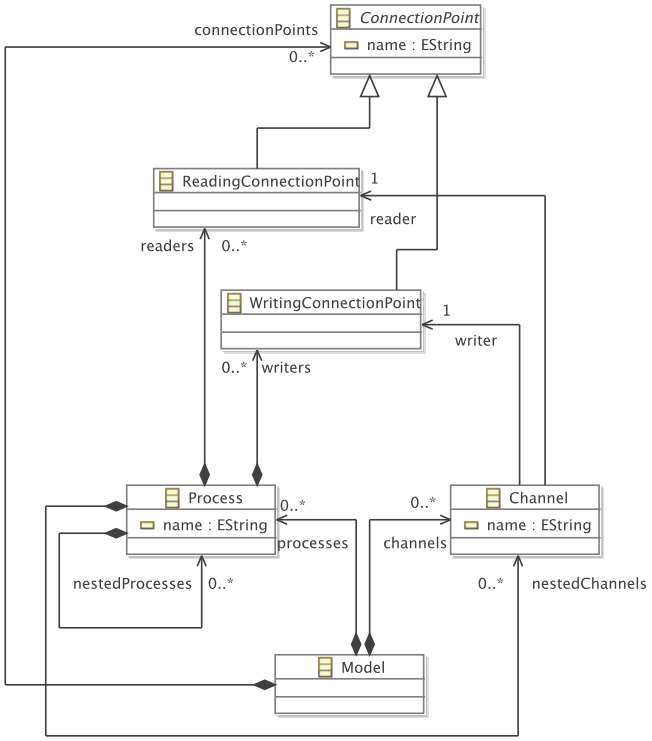


Figure B.11: The process-oriented metamodel after the final iteration.

As shown in Listing B.5, the "@gmf.compartment" annotation was added to the `nestedProcess` to indicate that processes can be placed inside other processes in the graphical editor.

```
1  @gmf.diagram
2  class Model {
3    val Process[*] processes;
4    val Channel[*] channels;
5    val ConnectionPoint[*] connectionPoints;
6  }
7
```

```
8  @gmf.node(label="name")
9  class Process {
10   attr String name;
11
12   @gmf.compartment
13   val Process[*] nestedProcesses;
14   val Channel[*] nestedChannels;
15
16   @gmf.affixed
17   ref ReadingConnectionPoint[*] readers;
18
19   @gmf.affixed
20   ref WritingConnectionPoint[*] writers;
21 }
22
23
24 @gmf.link(source="reader", target="writer", label="name", incoming="
       true")
25 class Channel {
26  attr String name;
27   ref ReadingConnectionPoint[1] reader;
28   ref WritingConnectionPoint[1] writer;
29 }
30
31 @gmf.node(label="name", label.placement="external", label.icon="false",
        figure="ellipse", size="15,15")
32 abstract class ConnectionPoint {
33   attr String name;
34 }
35
36 class ReadingConnectionPoint extends ConnectionPoint {}
37
38 @gmf.node(color="0,0,0")
39 class WritingConnectionPoint extends ConnectionPoint {}
```

Listing B.5: The annotated process-oriented metamodel after the final iteration

EuGENia was invoked on the annotated metamodel to produce the final version of the graphical editor. An additional model was constructed to check the nesting of processes, and is shown in Figure B.12. Because the changes made to the metamodel in this iteration involved only adding new features, no migration of existing models was necessary.

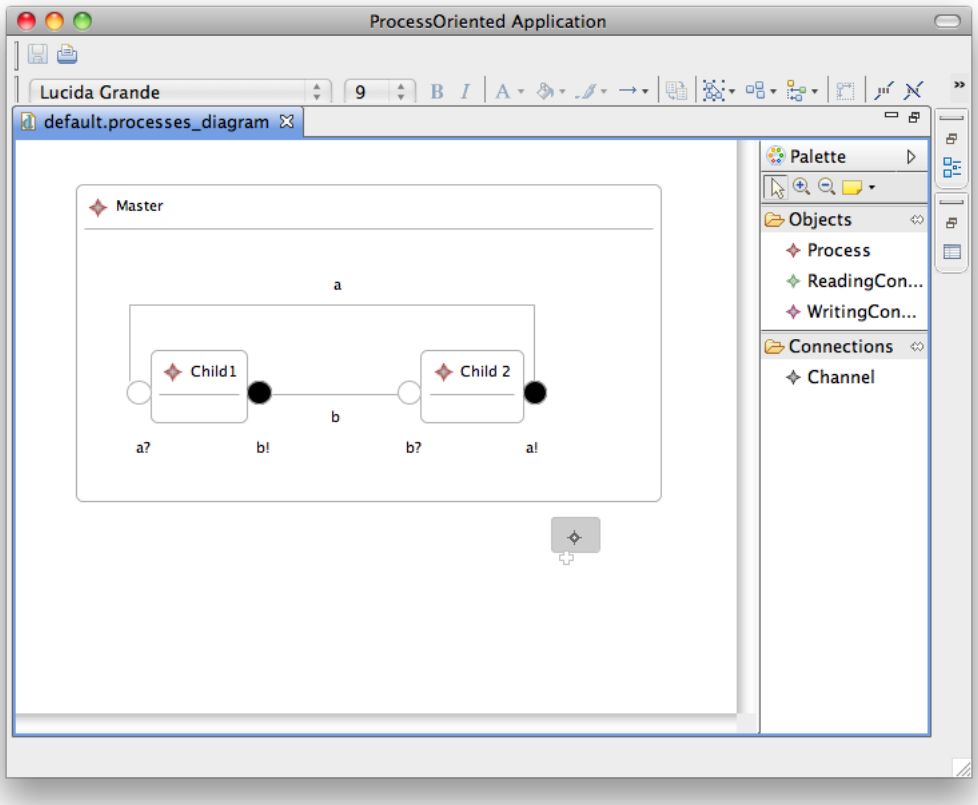


Figure B.12: Exemplar diagram after the final iteration.

B.7 Summary

This appendix has described the way in which a graphical editor for process-oriented programs was designed and implemented using an iterative style of development. A metamodel was used to capture the key concepts of the domain, and to generate code for a graphical editor. Each iteration involved changing the metamodel either to correct unintended behaviour in the editor (iterations 3 and 5), to facilitate interoperability with other tools (iteration 2) or to add new features (iterations 1, 4 and 6). The metamodel changes described in the fifth iteration are used for evaluation of the thesis research in Section 6.1.

Bibliography

[37-Signals 2008] 37-Signals. Ruby on Rails [online]. [Accessed 30 June 2008] Available at: `http://www.rubyonrails.org/`, 2008.

[Ackoff 1962] Russell L. Ackoff. *Scientific Method: Optimizing Applied Research Decisions*. John Wiley and Sons, 1962.

[Aizenbud-Reshef *et al.* 2005] N. Aizenbud-Reshef, R.F. Paige, J. Rubin, Y. Shaham-Gafni, and D.S. Kolovos. Operational semantics for traceability. In *Proc. ECMDA-FA Workshop on Traceability*, pages 8–14, 2005.

[Alexander *et al.* 1977] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series)*. Oxford University Press, 1977.

[Álvarez *et al.* 2001] José Álvarez, Andy Evans, and Paul Sammut. MML and the metamodel architecture. In *Proc. Workshop on Transformation in UML*, 2001.

[Apostel 1960] Leo Apostel. Towards the formal study of models in the non-formal sciences. *Synthese*, 12:125–161, 1960.

[Arendt *et al.* 2009] Thorsten Arendt, Florian Mantz, Lars Schneider, and Gabriele Taentzer. Model refactoring in eclipse by LTK, EWL, and EMF Refactor: A case study. In *Proc. Joint MoDSE-MCCM Workshop*, 2009.

[ATLAS 2007] ATLAS. Atlas Transformation Language Project Website [online]. [Accessed 30 June 2008] Available at: `http://www.eclipse.org/m2m/atl/`, 2007.

[Backus 1978] John Backus. The history of FORTRAN I, II and III. *History of Programming Languages*, 1:165–180, 1978.

[Balazinska *et al.* 2000] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lagüe, and Kostas Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proc. Working Conference on Reverse Engineering*, pages 98–107. IEEE Computer Society, 2000.

[Banerjee *et al.* 1987] Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *Proc. Special Interest Group on Management of Data*, volume 16, pages 311–322. ACM, 1987.

[Beck & Cunningham 1989] Kent Beck and Ward Cunningham. Constructing abstractions for object-oriented applications. *Journal of Object Oriented Programming*, 2, 1989.

[Bézivin & Gerbé 2001] Jean Bézivin and Olivier Gerbé. Towards a precise definition of the OMG/MDA framework. In *Proc. ASE*, pages 273–280. IEEE Computer Society, 2001.

[Bézivin 2005] Jean Bézivin. On the unification power of models. *Software and System Modeling*, 4(2):171–188, 2005.

[Biermann *et al.* 2006] Enrico Biermann, Karsten Ehrig, Christian Köhler, Günter Kuhns, Gabriele Taentzer, and Eduard Weiss. Emf model refactoring based on graph transformation concepts. *ECEASST*, 3, 2006.

[Bloch 2005] Joshua Bloch. How to design a good API and why it matters [online]. Keynote address to the LCSD Workshop at OOPSLA, October 2005, San Diego, United States of America. [Accessed 23 July 2009] Available at: `http://lcsd05.cs.tamu.edu/slides/keynote.pdf`, 2005.

[Bohner 2002] Shawn A. Bohner. Software change impacts - an evolving perspective. In *Proc. International Conference on Software Maintenance (ICSM)*, pages 263–272. IEEE Computer Society, 2002.

[Bosch 1998] Jan Bosch. Design patterns as language constructs. *Journal of Object Oriented Programming*, 11(2):18–32, 1998.

[Briand *et al.* 2003] Lionel C. Briand, Yvan Labiche, and L. O'Sullivan. Impact analysis and change management of uml models. In *Proc. International Conference on Software Maintenance (ICSM)*, pages 256–265. IEEE Computer Society, 2003.

[Brooks 1986] Frederick P. Brooks. No silver bullet – essence and accident in software engineering (invited paper). In *Proc. International Federation for Information Processing*, pages 1069–1076, 1986.

[Brown *et al.* 1998] William J. Brown, Raphael C. Malveau, Hays W. McCormick III, and Thomas J. Mowbray. *Anti Patterns*. Wiley, 1998.

[Cervelle *et al.* 2006] Julien Cervelle, Rémi Forax, and Gilles Roussel. Tatoo: an innovative parser generator. In *Principles and Practice of Programming in Java*, pages 13–20. ACM, 2006.

[Ceteva 2008] Ceteva. XMF – the extensible programming language [online]. [Accessed 30 June 2008] Available at: `http://www.ceteva.com/xmf.html`, 2008.

[Chen & Chou 1999] J.Y.J. Chen and S.C. Chou. Consistency management in a process environment. *Systems and Software*, 47(2-3):105–110, 1999.

[Cicchetti *et al.* 2008] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Automating co-evolution in model-driven engineering. In *Proc. EDOC*, pages 222–231. IEEE Computer Society, 2008.

[Clark *et al.* 2008] Tony Clark, Paul Sammut, and James Willians. Superlanguages: Developing languages and applications with XMF [online]. [Accessed 30 June 2008] Available at: `http://www.ceteva.com/docs/Superlanguages.pdf`, 2008.

[Cleland-Huang *et al.* 2003] Jane Cleland-Huang, Carl K. Chang, and Mark Christensen. Event-based traceability for managing evolutionary change. *IEEE Transactions on Software Engineering*, 29(9):796–810, 2003.

[Costa & Silva 2007] M. Costa and A.R. da Silva. RT-MDD framework – a practical approach. In *Proc. ECMDA-FA Workshop on Traceability*, pages 17–26, 2007.

[Czarnecki & Helsen 2006] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646, 2006.

[Deursen *et al.* 2000] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[Deursen *et al.* 2007] Arie van Deursen, Eelco Visser, and Jos Warmer. Model-driven software evolution: A research agenda. In *Proc. Workshop on Model-Driven Software Evolution*, pages 41–49, 2007.

[Dig & Johnson 2006a] Danny Dig and Ralph Johnson. Automated upgrading of component-based applications. In *OOPSLA Companion*, pages 675–676, 2006.

[Dig & Johnson 2006b] Danny Dig and Ralph Johnson. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance and Evolution*, 18(2):83–107, 2006.

[Dig *et al.* 2006] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automated detection of refactorings in evolving components. In *Proc. European Conference on Object-Oriented Programming*, volume 4067 of *LNCS*, pages 404–428. Springer, 2006.

[Dig *et al.* 2007] Danny Dig, Kashif Manzoor, Ralph Johnson, and Tien N. Nguyen. Refactoring-aware configuration management for object-oriented programs. In *Proc. International Conference on Software Engineering*, pages 427–436. IEEE Computer Society, 2007.

[Dig 2007] Daniel Dig. *Automated Upgrading of Component-Based Applications*. PhD thesis, University of Illinois at Urbana-Champaign, USA, 2007.

[Dmitriev 2004] Sergey Dmitriev. Language oriented programming: The next programming paradigm. *JetBrains onBoard [online]*, 1, 2004. [Accessed 30 June 2008] Available at: `http://www.onboard.jetbrains.com/is1/articles/04/10/lop/`.

[Drivalos *et al.* 2008] Nicholas Drivalos, Richard F. Paige, Kiran J. Fernandes, and Dimitrios S. Kolovos. Towards rigorously defined model-to-model traceability. In *Proc. European Conference on the Model Driven Architecture Workshop on Traceability*, 2008.

[Ducasse *et al.* 1999] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Proc. International Conference on Software Maintenance*, pages 109–118. IEEE Computer Society, 1999.

[Eclipse 2008a] Eclipse. Eclipse Modeling Framework project [online]. [Accessed 22 January 2009] Available at: `http://www.eclipse.org/modeling/emf/`, 2008.

[Eclipse 2008b] Eclipse. Eclipse project [online]. [Accessed 20 January 2009] Available at: `http://www.eclipse.org`, 2008.

[Eclipse 2008c] Eclipse. Epsilon home page [online]. [Accessed 30 June 2008] Available at: `http://www.eclipse.org/gmt/epsilon/`, 2008.

[Eclipse 2008d] Eclipse. Generative Modelling Technologies project [online]. [Accessed 30 June 2008] Available at: `http://www.eclipse.org/gmt`, 2008.

[Eclipse 2009a] Eclipse. Model Development Tools project [online]. [Accessed 6 January 2008] Available at: `http://www.eclipse.org/modeling/mdt/`, 2009.

[Eclipse 2009b] Eclipse. UML2 Model Development Tools project [online]. [Accessed 7 September 2009] Available at: `http://www.eclipse.org/modeling/mdt/uml2`, 2009.

[Eclipse 2010] Eclipse. Connected Data Objects Model Repository Project Website [online]. [Accessed 15 February 2010] Available at: `http://www.eclipse.org/modeling/emf/?project=cdo#cdo`, 2010.

[Edelweiss & Freitas Moreira 2005] Nina Edelweiss and Álvaro Freitas Moreira. Temporal and versioning model for schema evolution in object-oriented databases. *Data & Knowledge Engineering*, 53(2):99–128, 2005.

[Elmasri & Navathe 2006] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley Longman, 2006.

[Erlikh 2000] Len Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.

[Evans 2004] E. Evans. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Addison-Wesley, Boston, MA, USA, 2004.

[Feathers 2004] Michael C. Feathers. *Working Effectively with Legacy Code*. Prentice Hall, 2004.

[Ferrandina *et al.* 1995] Fabrizio Ferrandina, Thorsten Meyer, Roberto Zicari, Guy Ferran, and Joëlle Madec. Schema and database evolution in the O2 object database system. In *Very Large Data Bases*, pages 170–181. Morgan Kaufmann, 1995.

[Fowler 1999] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[Fowler 2002] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[Fowler 2005] Martin Fowler. Language workbenches: The killer-app for domain specific languages? [online]. [Accessed 30 June 2008] Available at: `http://www.martinfowler.com/articles/languageWorkbench.html`, 2005.

[Fowler 2010] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 2010.

[Frankel 2002] David Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley, 2002.

[Frenzel 2006] Leif Frenzel. The language toolkit: An API for automated refactorings in eclipse-based IDEs [online]. [Accessed 02 August 2010] Available at: `http://www.eclipse.org/articles/Article-LTK/ltk.html`, 2006.

[Fritzsche *et al.* 2008] M. Fritzsche, J. Johannes, S. Zschaler, A. Zherebtsov, and A. Terekhov. Application of tracing techniques in Model-Driven Performance Engineering. In *Proc. ECMDA Traceability Workshop (ECMDA-TW)*, pages 111–120, 2008.

[Fuhrer *et al.* 2007] Robert M. Fuhrer, Adam Kiezun, and Markus Keller. Refactoring in the Eclipse JDT: Past, present, and future. In *Proc. Workshop on Refactoring Tools*, 2007.

[Gamma *et al.* 1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[Garcés *et al.* 2009] Kelly Garcés, Frédéric Jouault, Pierre Cointe, and Jean Bézivin. Managing model adaptation by precise detection of metamodel changes. In *Proc. ECMDA-FA*, volume 5562 of *LNCS*, pages 34–49. Springer, 2009.

[Gosling *et al.* 2005] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java^{TM} Language Specification*. Addison-Wesley, Boston, MA, USA, 2005.

[Graham 1993] Paul Graham. *On Lisp: Advanced Techniques for Common Lisp*. Prentice-Hall, 1993.

[Greenfield *et al.* 2004] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004.

[Gronback 2006] Richard Gronback. Introduction to the Eclipse Graphical Modeling Framework. In *Proc. EclipseCon*, Santa Clara, California, 2006.

[Gronback 2009] R.C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 2009.

[Gruschko *et al.* 2007] Boris Gruschko, Dimitrios S. Kolovos, and Richard F. Paige. Towards synchronizing models with evolving metamodels. In *Proc. Workshop on Model-Driven Software Evolution*, 2007.

[Guerrini *et al.* 2005] Giovanna Guerrini, Marco Mesiti, and Daniele Rossi. Impact of XML schema evolution on valid documents. In *Proc. Workshop on Web Information and Data Management*, pages 39–44, 2005.

[Hearnden *et al.* 2006] David Hearnden, Michael Lawley, and Kerry Raymond. Incremental model transformation for the evolution of model-driven systems. In *Model Driven Engineering Languages and Systems*, volume 4199 of *LNCS*, pages 321–335. Springer, 2006.

[Heidenreich *et al.* 2009] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and refinement of textual syntax for models. In *Proc. ECMDA-FA*, volume 5562 of *LNCS*, pages 114–129. Springer, 2009.

[Herrmannsdoerfer *et al.* 2008] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Juergens. Automatability of coupled evolution of metamodels and models in practice. In *Proc. International Conference on Model Driven Engineering Languages and Systems*, volume 5301 of *LNCS*, pages 645–659. Springer, 2008.

[Herrmannsdoerfer *et al.* 2009a] M. Herrmannsdoerfer, D. Ratiu, and G. Wachsmuth. Language evolution in practice. In *Proc. SLE*, volume 5696 of *LNCS*, pages 3–22. Springer, 2009.

[Herrmannsdoerfer *et al.* 2009b] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Juergens. COPE - automating coupled evolution of metamodels and models. In *Proc. ECOOP*, volume 5653 of *LNCS*, pages 52–76. Springer, 2009.

[Hussey & Paternostro 2006] Kenn Hussey and Marcelo Paternostro. Advanced features of EMF. Tutorial at EclipseCon 2006, California, USA. [Accessed 07 September 2009] Available at: `http://www.eclipsecon.org/2006/Sub.do?id=171`, 2006.

[IBM 2005] IBM. Emfatic Language for EMF Development [online]. [Accessed 30 June 2008] Available at: `http://www.alphaworks.ibm.com/tech/emfatic`, 2005.

[INRIA 2007] INRIA. AMMA project page [online]. [Accessed 30 June 2008] Available at: `http://wiki.eclipse.org/AMMA`, 2007.

[IRISA 2007] IRISA. Sintaks. `http://www.kermeta.org/sintaks/`, 2007.

[ISO/IEC 1996] Information Technology ISO/IEC. Syntactic metalanguage – Extended BNF. ISO 14977:1996 International Standard, 1996.

[ISO/IEC 2002] Information Technology ISO/IEC. Z Formal Specification Notation – Syntax, Type System and Semantics. ISO 13568:2002 International Standard, 2002.

[Jackson 1995] M. Jackson. *Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices*. ACM Press, 1995.

[JetBrains 2008] JetBrains. MPS – Meta Programming System [online]. [Accessed 30 June 2008] Available at: `http://www.jetbrains.com/mps/index.html`, 2008.

[Jouault & Kurtev 2005] Frédéric Jouault and Ivan Kurtev. Transforming models with ATL. In *Proc. Satellite Events at the International Conference on Model Driven Engineering Languages and Systems*, volume 3844 of *LNCS*, pages 128–138. Springer, 2005.

[Jouault 2005] Frédéric Jouault. Loosely coupled traceability for ATL. In *Proc. ECMDA-FA Workshop on Traceability*, 2005.

[Kataoka *et al.* 2001] Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David Notkin. Automated support for program refactoring using invariants. In *Proc. International Conference on Software Maintenance*, pages 736–743. IEEE Computer Society, 2001.

[Kelly & Tolvanen 2008] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modelling.* Wiley, 2008.

[Kerievsky 2004] Joshua Kerievsky. *Refactoring to Patterns.* Addison-Wesley, 2004.

[Kleppe *et al.* 2003] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise.* Addison-Wesley, 2003.

[Klint *et al.* 2003] P. Klint, R. Lämmel, and C. Verhoef. Towards an engineering discipline for grammarware. *ACM Transactions on Software Engineering Methodology*, 14:331–380, 2003.

[Kolovos *et al.* 2006a] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Merging models with the epsilon merging language (eml). In *Proc. MoDELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 215–229. Springer, 2006.

[Kolovos *et al.* 2006b] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Model comparison: a foundation for model composition and model transformation testing. In *Proc. Workshop on Global Integrated Model Management*, pages 13–20, 2006.

[Kolovos *et al.* 2006c] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. The Epsilon Object Language (EOL). In *Proc. ECMDA-FA*, volume 4066 of *LNCS*, pages 128–142. Springer, 2006.

[Kolovos *et al.* 2007a] Dimitrios S. Kolovos, Richard F. Paige, Fiona A.C. Polack, and Louis M. Rose. Update transformations in the small with the Epsilon Wizard Language. *Journal of Object Technology*, 6(9):53–69, 2007.

[Kolovos *et al.* 2007b] Dimitrios S. Kolovos, Richard F. Paige, Louis M. Rose, and Fiona A.C. Polack. Bridging the Epsilon Wizard Language and the Eclipse Graphical Modeling Framework. In *Proc. Eclipse Summit*, Ludwigsburg, Germany, 2007.

[Kolovos *et al.* 2008a] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. The Epsilon Transformation Language. In *Proc. ICMT*, volume 5063 of *LNCS*, pages 46–60. Springer, 2008.

[Kolovos *et al.* 2008b] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. On the evolution of OCL for capturing structural constraints in modelling languages. In *Proc. Workshop on Rigorous Methods for Software Construction and Analysis*, 2008.

[Kolovos *et al.* 2008c] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Scalability : The holy grail of model driven engineering. In *Proc. Workshop on Challenges in Model Driven Engineering*, 2008.

[Kolovos *et al.* 2009] Dimitrios S. Kolovos, Richard F. Paige, and Louis M. Rose. EuGENia: GMF for mortals. Long talk at Eclipse Summit Europe, October 2009, Ludwigsburg, Germany. Available at: `https://www.eclipsecon.org/submissions/ese2009/view_talk.php?id=979` [Accessed 12 April 2010], 2009.

[Kolovos 2009] Dimitrios S. Kolovos. *An Extensible Platform for Specification of Integrated Languages for Model Management.* PhD thesis, University of York, United Kingdom, 2009.

[Kramer 2001] Diane Kramer. XEM: XML Evolution Management. Master's thesis, Worcester Polytechnic Institute, MA, USA, 2001.

[Kurtev 2004] Ivan Kurtev. *Adaptability of Model Transformations.* PhD thesis, University of Twente, Netherlands, 2004.

[Lago *et al.* 2009] Patricia Lago, Henry Muccini, and Hans van Vliet. A scoped approach to traceability management. *Systems and Software*, 82(1):168–182, 2009.

[Lämmel & Verhoef 2001] R. Lämmel and C. Verhoef. Semi-automatic grammar recovery. *Software - Practice and Experience*, 31(15):1395–1438, 2001.

[Lämmel 2001] R. Lämmel. Grammar adaptation. In *Proc. Formal Methods for Increasing Software Productivity (FME), International Symposium of Formal Methods Europe*, volume 2021 of *LNCS*, pages 550–570. Springer, 2001.

[Lämmel 2002] R. Lämmel. Towards generic refactoring. In *Proc. ACM SIGPLAN Workshop on Rule-Based Programming*, pages 15–28. ACM, 2002.

[Lehman 1969] Meir M. Lehman. The programming process. Technical report, IBM Res. Rep. RC 2722, 1969.

[Lerner 2000] Barbara Staudt Lerner. A model for compound type changes encountered in schema evolution. *ACM Transactions on Database Systems*, 25(1):83–127, 2000.

[Mäder *et al.* 2008] P. Mäder, O. Gotel, and I. Philippow. Rule-based maintenance of post-requirements traceability relations. In *Proc. IEEE International Requirements Engineering Conference (RE)*, pages 23–32, 2008.

[Martin & Martin 2006] R.C. Martin and M. Martin. *Agile Principles, Patterns, and Practices in C#*. Prentice Hall, Upper Saddle River, NJ, USA, 2006.

[McCarthy 1978] John McCarthy. History of Lisp. *History of Programming Languages*, 1:217–223, 1978.

[McNeile 2003] Ashley McNeile. MDA: The vision with the hole? [Accessed 30 June 2008] Available at: `http://www.metamaxim.com/download/documents/MDAv1.pdf`, 2003.

[Mellor & Balcer 2002] Stephen J. Mellor and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman, 2002.

[Melnik 2004] Sergey Melnik. *Generic Model Management: Concepts and Algorithms*. PhD thesis, University of Leipzig, Germany, 2004.

[Mens & Demeyer 2007] Tom Mens and Serge Demeyer. *Software Evolution*. Springer-Verlag, 2007.

[Mens & Tourwé 2004] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[Mens *et al.* 2007] Tom Mens, Gabriele Taentzer, and Dirk Müller. Challenges in model refactoring. In *Proc. Workshop on Object-Oriented Reengineering*, 2007.

[Merriam-Webster 2010] Merriam-Webster. Definition of Nuclear Family. `http://www.merriam-webster.com/dictionary/nuclear%20family`, 2010.

[Moad 1990] J Moad. Maintaining the competitive edge. *Datamation*, 36(4):61–66, 1990.

[Moha *et al.* 2009] Naouel Moha, Vincent Mahé, Olivier Barais, and Jean-Marc Jézéquel. Generic model refactorings. In *Proc. MoDELS*, volume 5795 of *LNCS*, pages 628–643. Springer, 2009.

[Muller & Hassenforder 2005] Pierre-Alain Muller and Michel Hassenforder. HUTN as a Bridge between ModelWare and GrammarWare. In *Proc. Workshop in Software Modelling Engineering*, 2005.

[Nentwich *et al.* 2003] C. Nentwich, W. Emmerich, A. Finkelstein, and E. Ellmer. Flexible consistency checking. *ACM Transactions on Software Engineering and Methodology*, 12(1):28–63, 2003.

[Nguyen *et al.* 2005] Tien Nhut Nguyen, Cheng Thao, and Ethan V. Munson. On product versioning for hypertexts. In *Proc. International Workshop on Software Configuration Management (SCM)*, pages 113–132. ACM, 2005.

[Oldevik *et al.* 2005] Jon Oldevik, Tor Neple, Roy Grønmo, Jan Øyvind Aagedal, and Arne-Jørgen Berre. Toward standardised model to text transformations. In *Proc. ECMDA-FA*, volume 3748 of *LNCS*, pages 239–253. Springer, 2005.

[Olsen & Oldevik 2007] Gøran K. Olsen and Jon Oldevik. Scenarios of traceability in model to text transformations. In *Proc. ECMDA-FA*, volume 4530 of *Lecture Notes in Computer Science*, pages 144–156. Springer, 2007.

[OMG 2001] OMG. Unified Modelling Language 1.4 Specification [online]. [Accessed 15 September 2008] Available at: `http://www.omg.org/spec/UML/1.4/`, 2001.

[OMG 2004] OMG. Human-Usable Textual Notation 1.0 Specification [online]. [Accessed 30 June 2008] Available at: `http://www.omg.org/technology/documents/formal/hutn.htm`, 2004.

[OMG 2005] OMG. MOF QVT Final Adopted Specication [online]. [Accessed 22 July 2009] Available at: `www.omg.org/docs/ptc/05-11-01.pdf`, 2005.

[OMG 2006] OMG. Object Constraint Language 2.0 Specification [online]. [Accessed 30 June 2008] Available at: `http://www.omg.org/technology/documents/formal/ocl.htm`, 2006.

[OMG 2007a] OMG. Unified Modelling Language 2.1.2 Specification [online]. [Accessed 30 June 2008] Available at: http://www.omg.org/spec/UML/2.1.2/, 2007.

[OMG 2007b] OMG. Unified Modelling Language 2.2 Specification [online]. [Accessed 5 March 2010] Available at: http://www.omg.org/spec/UML/2.2/, 2007.

[OMG 2007c] OMG. XML Metadata Interchange 2.1.1 Specification [online]. [Accessed 30 June 2008] Available at: http://www.omg.org/technology/documents/formal/xmi.htm, 2007.

[OMG 2008a] OMG. Meta-Object Facility [online]. [Accessed 30 June 2008] Available at: http://www.omg.org/mof, 2008.

[OMG 2008b] OMG. Model Driven Architecture [online]. [Accessed 30 June 2008] Available at: http://www.omg.org/mda/, 2008.

[OMG 2008c] OMG. Object Management Group home page [online]. [Accessed 30 June 2008] Available at: http://www.omg.org, 2008.

[Opdyke 1992] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, IL, USA, 1992.

[openArchitectureWare 2007] openArchitectureWare. openArchitectureWare Project Website [online]. [Accessed 30 June 2008] Available at: http://www.eclipse.org/gmt/oaw/, 2007.

[openArchitectureWare 2008] openArchitectureWare. XPand Language Reference [online]. [Accessed 18 August 2010] Available at: http://wiki.eclipse.org/AMMA, 2008.

[Paige *et al.* 2007] Richard F. Paige, Phillip J. Brooke, and Jonathan S. Ostroff. Metamodel-based model conformance and multiview consistency checking. *ACM Transactions on Software Engineering and Methodology*, 16(3), 2007.

[Paige *et al.* 2009] Richard F. Paige, Louis M. Rose, Xiaocheng Ge, Dimitrios S. Kolovos, and Phillip J. Brooke. FPTC: Automated safety analysis for domain-specific languages. In *MoDELS Workshops and Symposia*, volume 5421 of *Lecture Notes in Computer Science*, pages 229–242. Springer, 2009.

[Parr 2007] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers, 2007.

[Patrascoiu & Rodgers 2004] Octavian Patrascoiu and Peter Rodgers. Embedding OCL expressions in YATL. In *Proc. OCL and Model-Driven Engineering Workshop*, 2004.

[Pilgrim *et al.* 2008] Jens von Pilgrim, Bert Vanhooff, Immo Schulz-Gerlach, and Yolande Berbers. Constructing and visualizing transformation chains. In *Proc. European Conference on the Model Driven Architecture – Foundations and Applications*, volume 5095 of *LNCS*, pages 17–32. Springer, 2008.

[Pizka & Jürgens 2007] M. Pizka and E. Jürgens. Automating language evolution. In *Proc. Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 305–315. IEEE Computer Society, 2007.

[Porres 2003] Ivan Porres. Model refactorings as rule-based update transformations. In *Proc. UML*, volume 2863 of *LNCS*, pages 159–174. Springer, 2003.

[Ramil & Lehman 2000] Juan F. Ramil and Meir M. Lehman. Cost estimation and evolvability monitoring for software evolution processes. In *Proc. Workshop on Empirical Studies of Software Maintenance*, 2000.

[Ráth *et al.* 2008] István Ráth, Gábor Bergmann, András Ökrös, and Dániel Varró. Live model transformations driven by incremental pattern matching. In *Proc. ICMT*, volume 5063 of *LNCS*, pages 107–121. Springer, 2008.

[Rising 2001] Linda Rising, editor. *Design patterns in communications software*. Cambridge University Press, 2001.

[Rose *et al.* 2008] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona A.C. Polack. The Epsilon Generation Language. In *Proc. European Conference on Model Driven Architecture – Foundations and Applications*, volume 5095 of *LNCS*, pages 1–16. Springer, 2008.

[Rose *et al.* 2009a] Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Enhanced automation for managing model and metamodel inconsistency. In *Proc. ASE*, pages 545–549. ACM Press, 2009.

[Rose *et al.* 2009b] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona A.C. Polack. An analysis of approaches to model migration. In *Proc. Joint MoDSE-MCCM Workshop*, 2009.

[Rose *et al.* 2010a] Louis M. Rose, Markus Herrmannsdoerfer, James R. Williams, Dimitrios S. Kolovos, Kelly Garcés, Richard F. Paige, and

Fiona A.C. Polack. A comparison of model migration tools. In *Proc. MoDELS*, volume TBC of *Lecture Notes in Computer Science*, page TBC. Springer, 2010.

[Rose *et al.* 2010b] Louis M. Rose, Dimitrios S. Kolovos, Nicholas Drivalos, James. R. Williams, Richard F. Paige, Fiona A.C. Polack, and Kiran J. Fernandes. Concordance: An efficient framework for managing model integrity [submitted to]. In *Proc. European Conference on Modelling Foundations and Applications*, volume 6138 of *LNCS*, pages 62–73. Springer, 2010.

[Rose *et al.* 2010c] Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Migrating activity diagrams with Epsilon Flock. In *Proc. TTC*, 2010.

[Rose *et al.* 2010d] Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Model migration case. In *Proc. TTC*, 2010.

[Rose *et al.* 2010e] Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Model migration with Epsilon Flock. In *Proc. ICMT*, volume 6142 of *Lecture Notes in Computer Science*, pages 184–198. Springer, 2010.

[Selic 2003] Bran Selic. The pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003.

[Selic 2005] Bran Selic. Whats new in UML 2.0? *IBM Rational software*, 2005.

[Sendall & Kozaczynski 2003] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20:42–45, 2003.

[Sjøberg 1993] Dag I.K. Sjøberg. Quantifying schema evolution. *Information & Software Technology*, 35(1):35–44, 1993.

[Sommerville 2006] Ian Sommerville. *Software Engineering*. Addison-Wesley Longman, 2006.

[Sprinkle & Karsai 2004] Jonathan Sprinkle and Gábor Karsai. A domain-specific visual language for domain model evolution. *Journal of Visual Languages and Computing*, 15(3-4):291–307, 2004.

[Sprinkle 2003] Jonathan Sprinkle. *Metamodel Driven Model Migration*. PhD thesis, Vanderbilt University, TN, USA, 2003.

[Sprinkle 2008] Jonathan Sprinkle. *Difference Representation and Conflict Management in Model-Driven Engineering*. PhD thesis, Universita' degli Studi dell'Aquila, L'Aquila, Italy, 2008.

[Stahl *et al.* 2006] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, 2006.

[Starfield *et al.* 1990] M. Starfield, K.A. Smith, and A.L. Bleloch. *How to model it: Problem Solving for the Computer Age*. McGraw-Hill, 1990.

[Steinberg *et al.* 2008] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2008.

[Su *et al.* 2001] Hong Su, Diane Kramer, Li Chen, Kajal T. Claypool, and Elke A. Rundensteiner. XEM: Managing the evolution of XML documents. In *Proc. Workshop on Research Issues in Data Engineering*, pages 103–110, 2001.

[Tratt 2008] Laurence Tratt. A change propagating model transformation language. *Journal of Object Technology*, 7(3):107–124, 2008.

[Varró & Balogh 2007] Dániel Varró and András Balogh. The model transformation language of the VIATRA2 framework. *Science of Computer Programming*, 68(3):187–207, 2007.

[Vries & Roddick 2004] Denise de Vries and John F. Roddick. Facilitating database attribute domain evolution using meso-data. In *Proc. Workshop on Evolution and Change in Data Management*, pages 429–440, 2004.

[W3C 2007a] W3C. W3C XML Schema 1.1 Specification [online]. [Accessed 30 June 2008] Available at: `http://www.w3.org/XML/Schema`, 2007.

[W3C 2007b] W3C. World Wide Web Consortium [online]. [Accessed 30 June 2008] Available at: `http://www.w3.org/`, 2007.

[Wachsmuth 2007] Guido Wachsmuth. Metamodel adaptation and model co-adaptation. In *Proc. ECOOP*, volume 4609 of *LNCS*, pages 600–624. Springer, 2007.

[Wallace 2005] Malcolm Wallace. Modular architectural representation and analysis of fault propagation and transformation. *Electronic Notes in Theoretical Computer Science*, 141(3):53–71, 2005.

[Ward 1994] Martin P. Ward. Language-oriented programming. *Software — Concepts and Tools*, 15(4):147–161, 1994.

[Watson 2008] Andrew Watson. A brief history of MDA. *Upgrade*, 9(2), 2008.

[Welch & Barnes 2005] Peter H. Welch and Fred R. M. Barnes. Communicating mobile processes. In *Proc. Symposium on the Occasion of 25 Years of Communicating Sequential Processes (CSP)*, volume 3525 of *LNCS*, pages 175–210. Springer, 2005.

[Winkler & Pilgrim 2009] Stefan Winkler and Jens von Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Software and Systems Modeling*, December 2009.