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-

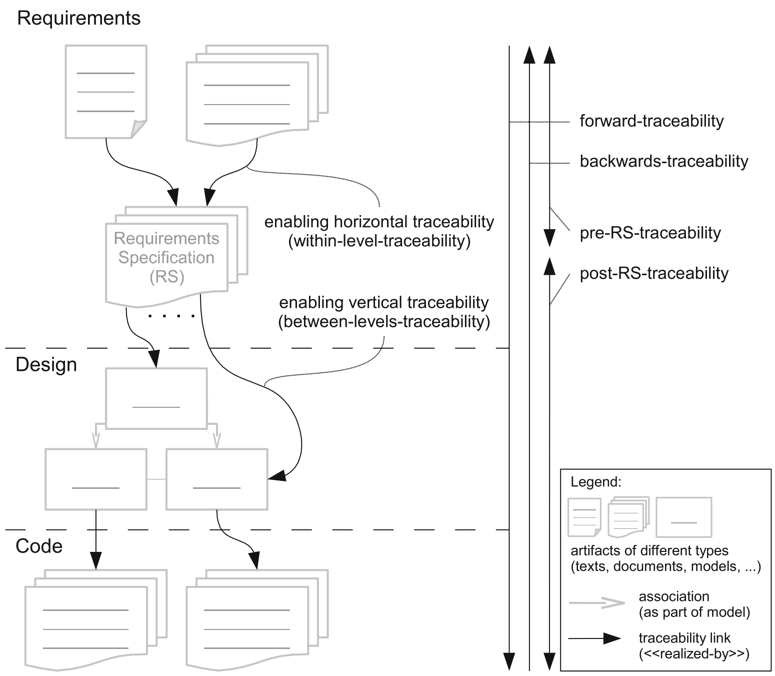


Figure 3.1: Categories of traceability link [Winkler & Pilgrim 2009].

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, proposed 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.

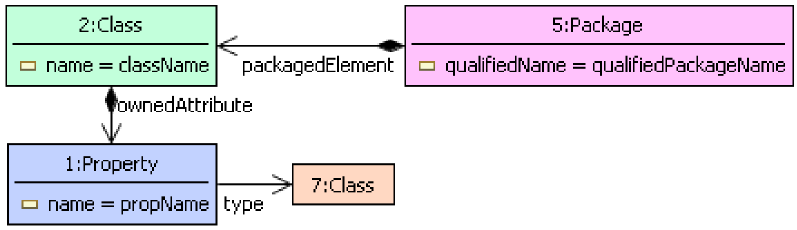
Abstraction is a fundamental benefit of MDE (Section 2.5.1). Defining a domain-specific language is one way in which abstraction can be realised for MDE (Section 2.4.1). In addition to tools for defining modelling languages, generating model editors and performing model transformation, model-driven development 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.

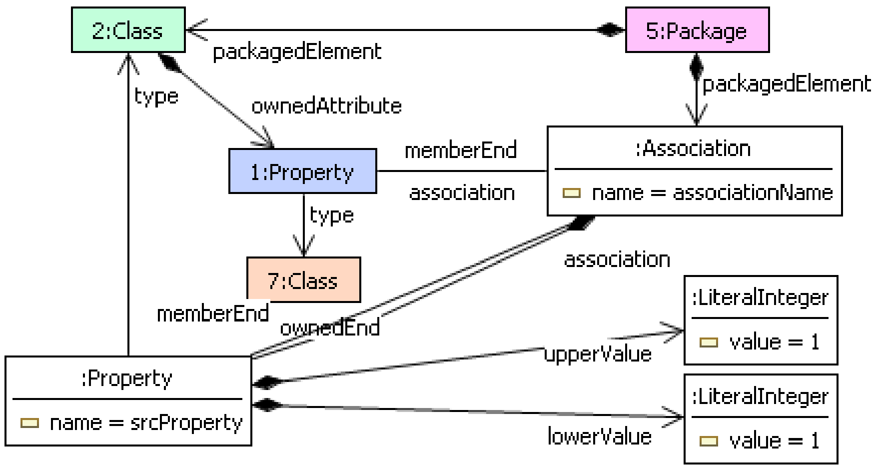
The Epsilon Wizard Language (EWL) [Kolovos *et al.* 2007a] is a model transformation language tailored for the specification of model refactorings. EWL is built atop Epsilon and its object language (EOL), which can query, update and navigate models represented in a diverse range of modelling technologies (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 4.3(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.

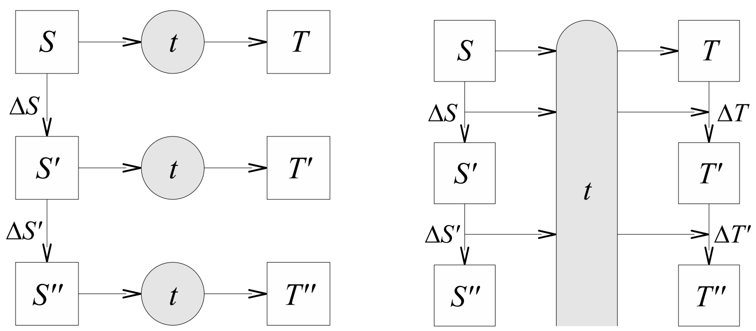
According to [Mens *et al.* 2007], "research in model refactoring is still in its infancy." Mens et al. identify formalisms for investigating the feasibility and scalability of model refactoring. In particular, Mens et al. suggest that meaning-preservation (an objective of refactoring, as discussed in 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 implementation.

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.* 2008b], 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 the 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], and use a co-evolution process similar to the one shown

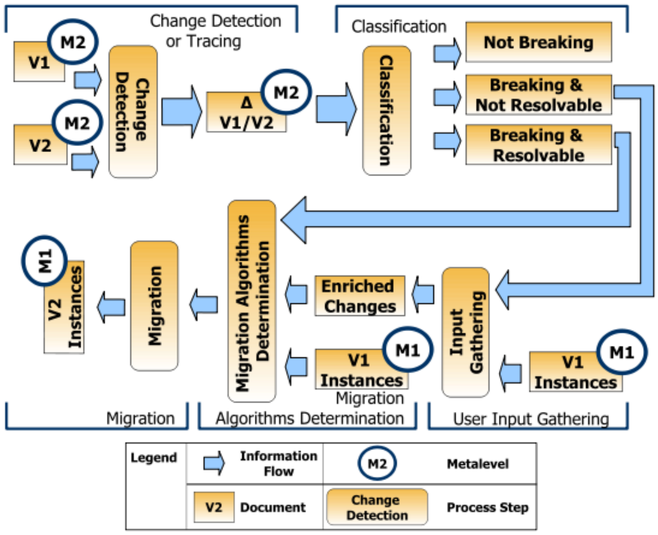


Figure 3.5: The co-evolution process described in [Gruschko *et al.* 2007].

in Figure 3.5. 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-

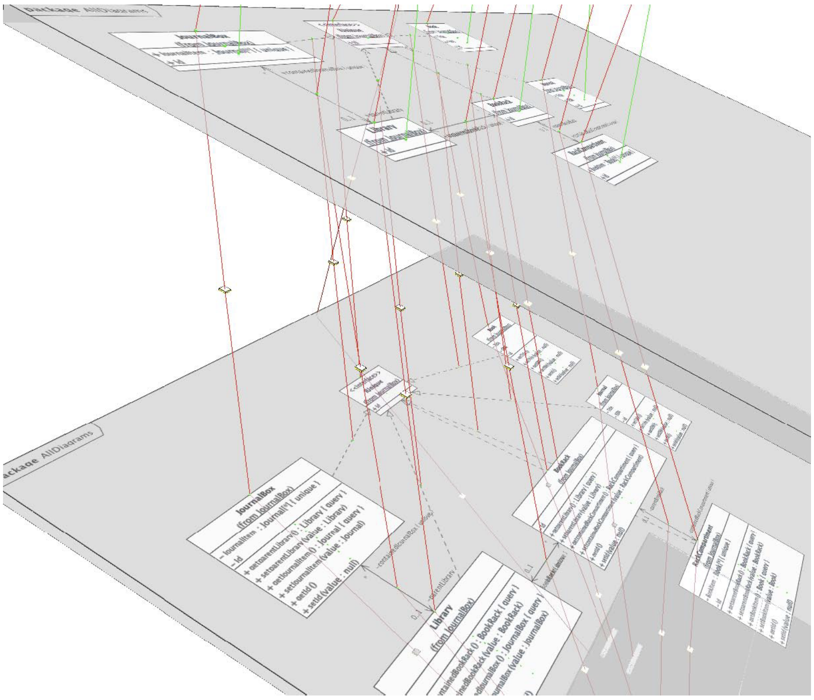


Figure 3.6: Visualising a transformation chain. Taken from [Pilgrim *et al.* 2008].

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. In addition, 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 review presented in Chapter 3 highlighted challenges for identifying and managing evolution in the context of MDE, and noted that little work has explored the way in which evolution occurs in practice. This chapter explores evolution in the context of MDE by identifying and analysing examples from software engineering projects developed in a model-driven manner. The analysis presented in this chapter facilitated the identification of requirements for the thesis research, which were approached by the work presented in Chapter 5.

Figure 4.1 summarises the objectives of this chapter. 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).

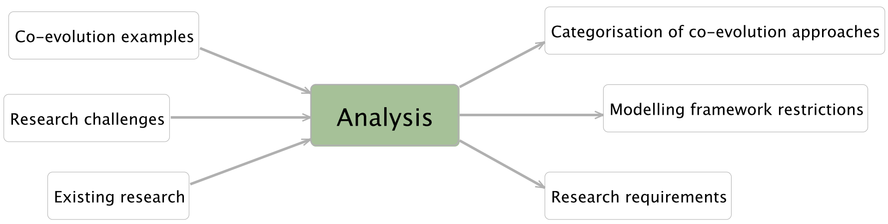


Figure 4.1: Analysis chapter overview.

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, contemporary MDE modelling frameworks such as the Eclipse Modeling Framework [Steinberg *et al.* 2008]. Therefore, the term *modelling framework* is used to mean contemporary 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).

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

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

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

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).

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 4.1.2 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 2008c] 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. Two zoos were considered (the Atlantic Zoo and the AtlantEcore Zoo[2]), but neither contained significant metamodel changes. Those changes that were made involved only renaming of meta-classes (trivial to migrate) or additive changes (which do not affect conformance, 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 the author developed an implementation of FPTC in Eclipse. The implementation includes an FPTC metamodel. More recent work with Philippa

[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/`

Conmy, a Research Associate in this department, has identified a significant flaw in the implementation, leading to changes to the metamodel. The metamodel changes affected the conformance of existing FPTC models. Conmy has made 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 do not 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 affected conformance. xText provides examples, 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 affected the conformance of existing 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). **Suitability for study:** Suitable for studying co-evolution. Unsuitable for studying synchronisation.

Summary of selection

Of the eight projects considered, three (FPTC, xText and GMF) fulfilled all of the mandatory requirements for studying co-evolution. No projects fulfilled all of the mandatory requirements for studying synchronisation. FPTC

[4] Private communication, 2008.

and xText were used to perform the analysis described in the remainder of this chapter, along with examples taken from other sources. GMF provides perhaps the most comprehensive examples of co-evolution, as it includes several metamodels that have undergone two major and several minor revisions, several exemplar models that have been migrated, and reference migration strategies (written in Java). Rather than use GMF for analysis, it was instead reserved for evaluation of the thesis research (Chapter 6).

4.1.3 Other examples

Few MDE projects fulfilled all of the requirements for studying evolution, so additional data was sought from alternative sources. Examples were located from object-oriented systems – which have some similarities to systems developed using MDE – and via 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 often non-breaking

in modelling frameworks, which typically infer values for the inverse of a reference when required.

3. **Domain-specific refactorings** manage issues not relevant to metamodels, such as casting, defensive return values, and assertions.

The object-oriented refactorings that can be applied to metamodels provide examples of metamodel evolution and, in some cases, have the potential to affect conformance. For each refactoring that affected conformance, a migration strategy was deduced by the author using Fowler's description of each refactoring. An example of this process is now presented.

Figure 4.2 illustrates a refactoring that changes a reference object to a value object [Fowler 1999][pg183]. 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.

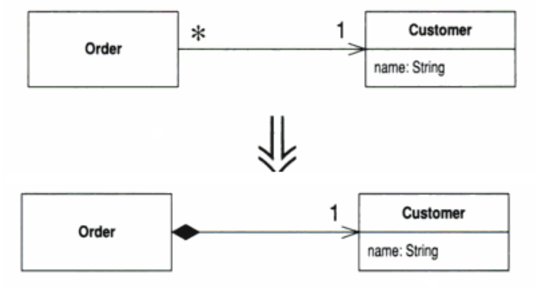


Figure 4.2: Refactoring a reference to a value. Taken from [Fowler 1999][pg183].

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 metamodels and affected conformance.

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

```

  delete c
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 – for addressing new requirements and facilitating interoperability with other systems – 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 graphical editor for process-oriented programs was developed with Adam Sampson, then a Research Associate at the University of Kent, and is described in Appendix [?]. Additionally, the feasibility of a tool for generating story-worlds for interactive narratives was investigated with Heather Barber, then a postdoctoral researcher in this 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

This section has described the identification of example data for analysing the way in which evolution is identified and managed in the context of MDE. Example data was sought from existing MDE projects, a related domain (refactoring of object-oriented systems) and collaborative work on MDE projects (with Sampson and Barber). Eight 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 for the evaluation presented 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, the remainder of the thesis focuses on model-metamodel co-evolution.

4.2 Analysing Existing Techniques

The examples of co-evolution identified in the previous section were analysed to identify and compare existing techniques for managing co-evolution. 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) . The work presented here was published in [Rose *et al.* 2009b, Rose *et al.* 2009a].

4.2.1 Modelling Framework Characteristics

Analysis of the co-evolution examples identified above highlighted characteristics of modern MDE modelling development environments that affect 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, 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 conform 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. Typically then, the metamodel user can only perform migration by editing the model directly, normally manipulating its underlying representation (e.g. XMI). Model editors and model management operations, which are ordinarily integral to MDE, cannot be used to manage models that do not conform to their metamodel and hence, cannot be used during model migration.

A further consequence of implicitly enforced conformance is that modelling tools must produce models that conforms to their metamodel, and therefore, model migration cannot be decomposed. Consequently, model migration cannot be performed by combining co-evolution techniques, because intermediate steps must produce conformant models.

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.* Analysis of the co-evolution examples identified in Section 4.1 highlighted several instances of user-driven co-evolution. Projects conducted in collaboration with Barber and with Sampson involved user-driven co-evolution, and two of the co-evolution examples taken from the xText project were managed in a user-driven manner. This section demonstrates user-driven co-evolution using a scenario similar to one observed during the collaboration with Barber.

In user-driven co-evolution, the metamodel user performs migration by loading their models to test conformance, and then reconciling 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.

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 [OMG 2007c], for example, is a standard model storage format and is regarded as sub-optimal for use by humans [OMG 2004]. Consequently, using a model storage format to perform model migration can be error-prone and tedious. When directly editing the underlying format of a model, reconciling conformance is often a slow and iterative process. For example, EMF [Steinberg *et al.* 2008], arguably the most widely used modelling framework, uses a multi-pass model parser and hence only reports one category of errors when a model cannot be loaded. 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 co-evolution. Firstly, the underlying model representation is unlikely to be optimised for human usability and hence user-driven co-evolution is error-prone and tedious. Secondly, although conformance can be affected when a new version of a metamodel is installed, 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. Some metamodels – such as UML, Ecore, and MOF – have many more users, and user-driven co-evolution would require repeated manual effort from each user. In spite of this, UML, for example, does not provide a strategy for migrating between versions of the specification, and users must infer the migration semantics from changes to the specification.

4.2.3 Developer-Driven Co-Evolution Approaches

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. Appendix A describes the way in which the co-evolution examples were explored with several existing developer-driven co-evolution approaches. From the analysis, three categories of developer-driven co-evolution approach were identified: *manual specification*, *operator-based* and *inference*. This categorisation has been published in [Rose *et al.* 2009b]. 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 inference 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 migration patterns in their chosen 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 might evolve the metamodel such that a non-containment reference becomes a containment reference and migrate 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 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 (EMF) [Steinberg *et al.* 2008].

The efficacy of an operator-based co-evolution approach depends heavily on the richness of its library of co-evolutionary operators. When no operator describes the required co-evolution pattern, the metamodel developer must use another approach for performing model migration. For instance, COPE allows migration to be specified manually with a general purpose programming language when no co-evolutionary operator is appropriate. (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).

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 pattern that a metamodel developer may wish to apply. 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[5], is shown below. To choose between operators, users can read descriptions (such as the one shown below) examine the source code of the operator, or try executing the operator (an undo command is provided).

> **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.

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

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

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.

Inference

In *inference* approaches, a migration strategy is derived from the evolved metamodel and the *metamodel history*. Inference approaches can be further categorised according to the type of metamodel history used. *Differencing* approaches compare and match the original and evolved metamodels, while *change recording* approaches use a record of primitive changes made to the original metamodel to produce the evolved metamodel. 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[6] to produce a model-to-model transformation from the difference model. [Cicchetti *et al.* 2008] and [Garcés *et al.* 2009] describe differencing-based inference 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, and [Méndez *et al.* 2010] suggest that change recording approach might be used to manage metamodel-transformation co-evolution.

Compared to manual specification and operator-based co-evolution approaches, inference approaches require 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, 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, an inference approach often requires guidance, because the metamodel changes alone do not provide enough information to correctly distinguish between feasible migration strategies. Existing inference approaches use heuristics to determine the most appropriate migration strategy. These heuristics sometimes lead to the selection of the wrong migration strategy.

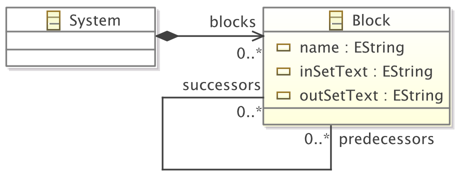
Because inference approaches use heuristics to select a migration strategy, it can sometimes be difficult to reason about which migration strategy will be selected. For domains where predictability, completeness and correctness are a primary concern (e.g. safety critical or security critical systems, or

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

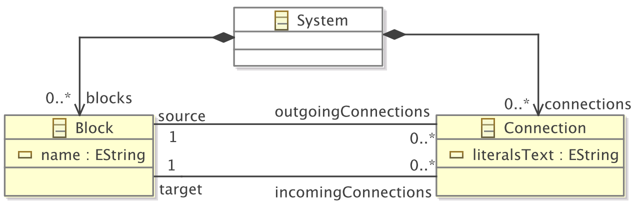
systems that must undergo certification with respect to a relevant standard), such approaches are unsuitable, and deterministic approaches that can be demonstrated to produce correct, predictable results will be required.

The two types of inference approach – differencing and change recording – are now compared, using an example of co-evolution, introduced below.

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[7]. Figure 4.3(a) 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.



(a) Original metamodel, prior to evolution



(b) Evolved metamodel with Connection metaclass

Figure 4.3: Metamodel evolution in the Epsilon FPTC tool. Taken from [Rose *et al.* 2009b].

Further analysis of the domain revealed that extra information about the relationship between `Blocks` was to be captured. The evolved metamodel is shown in Figure **??**. The `Connection` class is introduced to capture the extra information via its `literalsText` attribute. `Blocks` are no longer related directly to `Blocks`, instead they are related via an instance of the `Connection` class. The `incomingConnections` and `outgoingConnections` references of `Block` are used to relate `Blocks` to each other via an instance of `Connection`.

A model that conforms to the original metamodel (Figure 4.3(a)) might not conform to the evolved metamodel (Figure **??**). Below is a description of

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

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.

Using the example described above, differencing and change recording inference approaches are now compared.

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`, Change the type of feature `successors` from `Block` to `Connection`). The record of changes may be reduced to a normal form to remove redundancy, but doing so can erase useful information. In change recording, some types of metamodel evolution can be more easily recognised than with differencing. With change recording, renaming can be distinguished from a deletion followed by an addition. 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 **??**) could have been produced by changing the name and type of the `successors` reference (shown in Figure 4.3(a)). 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`. Clearly then, 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. [Cicchetti 2008] discusses this issue, and proposes a normal form 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(a)) and evolved (Figure **??**) 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 identification of user-driven co-evolution and to a categorisation of developer-driven 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 co-evolution). When no executable migration strategy is provided, models must be migrated by hand (user-driven co-evolution). Because modelling frameworks implicitly enforce conformance, user-driven co-evolution is performed by editing the underlying storage representation of models, which is error-prone and tedious.

User-driven co-evolution, which has not been explored in the literature, 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 co-evolution is required.

Existing techniques for performing developer-driven co-evolution have been compared and categorised. The categorisation highlights a trade-off between flexibility and effort for the metamodel-developer when choosing between categories of approach, as shown in Figure 4.4.

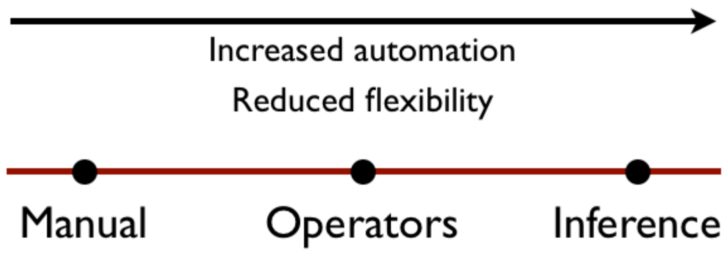


Figure 4.4: Spectrum of developer-driven co-evolution approaches

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, inference approaches seek to infer a migration strategy from a metamodel history and hence require less effort from the metamodel developer. However, an inference 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. [8]

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.*

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

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

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 inference 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 an inference 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 [**?**], 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.

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.

[Cicchetti 2008] Antonio Cicchetti. *Difference Representation and Conflict Management in Model-Driven Engineering*. PhD thesis, Universita' degli Studi dell'Aquila, L'Aquila, Italy, 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.

[Geiß & Kroll 2007] Rubino Geiß and Moritz Kroll. Grgen.net: A fast, expressive, and general purpose graph rewrite tool. In *Proc. Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 5088 of *Lecture Notes in Computer Science*, pages 568–569. Springer, 2007.

[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 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.

[Halstead 1977] Maurice H. Halstead. *Elements of Software Science*. Elsevier Science Inc., 1977.

[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.

[Herrmannsdoerfer *et al.* 2010] Markus Herrmannsdoerfer, Sander Vermolen, and Guido Wachsmuth. An extensive catalog of operators for the coupled evolution of metamodels and models. In *Proc. SLE*, volume TBC of *LNCS*, page TBC. Springer, 2010.

[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.

[Jurack & Mantz 2010] Stefan Jurack and Florian Mantz. Towards metamodel evolution of EMF models with Henshin. In *Proc. ME Workshop*, 2010.

[Kalnins *et al.* 2005] Audris Kalnins, Janis Barzdins, and Edgars Celms. Model transformation language MOLA. In *Proc. Model Driven Architecture, European MDA Workshops: Foundations and Applications MDAFA*, volume 3599 of *Lecture Notes in Computer Science*, pages 62–76. Springer, 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.

[Lara & Guerra 2010] Juan de Lara and Esther Guerra. Generic metamodelling with concepts, templates and mixin layers. In *Proc. MoDELS (1)*, volume 6394 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 2010.

[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.

[Méndez *et al.* 2010] David Méndez, Anne Etien, Alexis Muller, and Rubby Casallas. Towards transformation migration after metamodel evolution. In *Proc. ME Workshop*, 2010.

[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.

[Nickel *et al.* 2000] Ulrich Nickel, Jörg Niere, and Albert Zündorf. The FUJABA environment. In *Proc. International Conference on Software Engineering (ICSE)*, pages 742–745, New York, NY, USA, 2000. ACM.

[Northrop 2006] L. Northrop. Ultra-large scale systems: The software challenge of the future. Technical report, Carnegie Mellon, June 2006.

[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.

[RAE & BCS 2004] The RAE and The BCS. The challenges of complex IT projects. Technical report, The Royal Academy of Engineering, April 2004.

[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.* 2008a] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona A.C. Polack. Constructing models with the Human-Usable Textual Notation. In *Proc. International Conference on Model Driven Engineering Languages and Systems*, volume 5301 of *LNCS*, pages 249–263. Springer, 2008.

[Rose *et al.* 2008b] 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, Anne Etien, David Méndez, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Comparing model-metamodel and transformation-metamodel co-evolution. In *Proc. ME Workshop*, 2010.

[Rose *et al.* 2010b] 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.* 2010c] 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.* 2010d] 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.* 2010e] Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Model migration case. In *Proc. TTC*, 2010.

[Rose *et al.* 2010f] 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.

[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.

[Tisi *et al.* 2009] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. On the use of higher-order model transformations. In *Proc. ECMDA-FA*, volume 5562 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2009.

[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.