Evolution in Model-Driven Engineering

Louis M. Rose

May 17, 2010

Contents

1 Introduction **5**
- 1.1 Model-Driven Engineering . . . 5
- 1.2 Software Evolution . . . 5
- 1.3 Research Aim . . . 6
- 1.4 Research Method . . . 6

2 Background **7**
- 2.1 Modelling . . . 7
- 2.2 Model-Driven Engineering . . . 8
- 2.3 Related Areas . . . 8
- 2.4 Categories of Evolution in MDE . . . 8

3 Literature Review **9**
- 3.1 Software Evolution Theory . . . 9
- 3.2 Software Evolution in Practice . . . 13
- 3.3 Summary . . . 25

4 Analysis **27**
- 4.1 Locating Data . . . 27
- 4.2 Analysing Existing Techniques . . . 34
- 4.3 Requirements Identification . . . 43
- 4.4 Chapter Summary . . . 46

5 Implementation **47**
- 5.1 Metamodel-Independent Syntax . . . 47
- 5.2 Textual Modelling Notation . . . 52
- 5.3 Epsilon Flock . . . 61
- 5.4 Chapter Summary . . . 71

6 Evaluation **73**
- 6.1 Evaluation Measures . . . 73
- 6.2 Discussion . . . 90
- 6.3 Dissemination / Reception / ?? . . . 90

7 Conclusion **91**
- 7.1 Future Work . . . 91

A Experiments **93**
- A.1 Metamodel-Independent Change . . . 93

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 open research challenges, highlighting those to which this thesis contributes.

3.1 Software Evolution Theory

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 two research areas that relate to software evolution: refactoring and design patterns. The former concentrates on improving the structure of existing systems, while the latter is more often used when designing software. Both areas provide 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 examples of 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, for instance, is one way in which perfective evolution can be realised.

3.1.2 Refactoring

In [Mens & Tourwé 2004], Mens and Tourwé report "an urgent need for techniques that reduce software complexity by incrementally improving the internal software quality." Refactoring – "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] – is being applied to resolve this issue. 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 [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 describes a number of refactorings particular to the Ruby on Rails web framework [37-Signals 2008], such as "Skinny Controller, Fat Model" [1].

MOF [OMG 2008a] 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), refactoring 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 as it is 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, the author anticipates that refactoring catalogues, such as the one provided by [Fowler 1999], are likely to 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.

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

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 often 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 [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 [Fowler 1999].

[Alexander *et al.* 1977] first utilised design patterns when devising a pattern catalogue for town planning. [Beck & Cunningham 1989] later adapted Alexander's ideas 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, pg10]. [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]). Coplien notes that "patterns and anti-patterns are complementary" [Brown *et al.* 1998, pg13]; 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. This section reviews and analyses traceability literature, focussing on the relationship between traceability and software evolution.

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

Traceability is facilitated by *traceability links*, which document the dependencies, causalities and influences between artefacts. Traceability links are established by hand and by automated analysis of artefacts. In MDD 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.

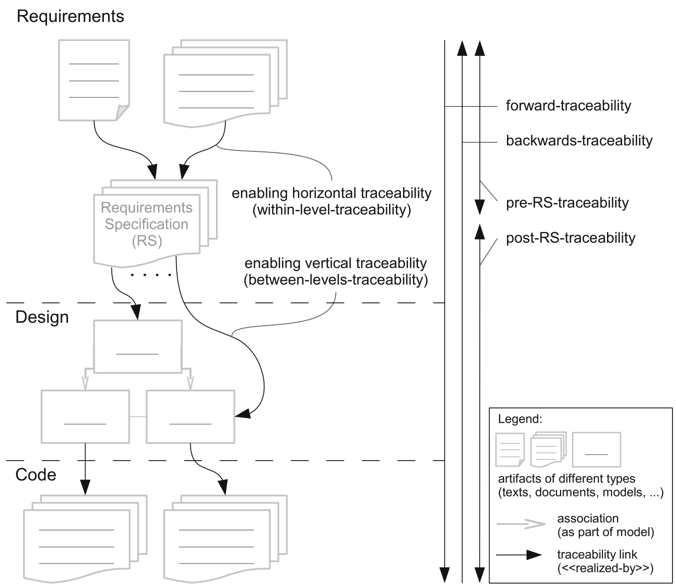


Figure 3.1: Categories of traceability link [Winkler & Pilgrim 2009].

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 to support manual 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-based approaches for impact analysis and change propagation in the context of model-driven development.

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, pg24], 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. As discussed in Chapter 4, pattern-based approaches make assumptions on the development environment and are effective only when they provide a rich yet navigable catalogue of patterns.

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 language, schema and model-driven development.

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 reductionj.

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]).

Some features of Lisp were devised by promoting Fortran List Processing Language (FLPL) design patterns to language constructs. 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). As FORTRAN programs could not express these semantics, McCarthy's new construct informed the design of Lisp.

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, pg30]. 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 (elaborated in Section 3.2.2) is used heavily in the XML schema evolution community, and was the sole strategy encountered. 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. In Section 3.2.2, 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 Schema 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. 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 utilises 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].

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]).

In [Vries & Roddick 2004], de Vries and Roddick 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] calls 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, pg334].

Between 2001 and 2005, Ralf Lämmel, co-author of [Klint *et al.* 2003], and his colleagues (at Vrije Universiteit, Amsterdam) 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 to both programs written in using that grammar and to tools that parse, generate or otherwise manipulate programs written in that grammar.

[Pizka & Jürgens 2007] recognises and seeks to address the challenge of grammar-program co-evolution. Pizka and Juergens believe that most domain-specific languages will 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. Lever 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).

This section focuses on the two types of co-evolution discussed in model-driven engineering literature. *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

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

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 of MDE development artefacts.

Model Refactoring

Refactoring (Section 3.1.2) is a perfective software evolution activity in which a system's quality is improved without changing its functional behaviour. Refactoring has been studied in the context of model-driven engineering because refactoring can be domain-specific (e.g. normalisation in relational databases). 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, 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.* 2007] (which mixes declarative and imperative parts).

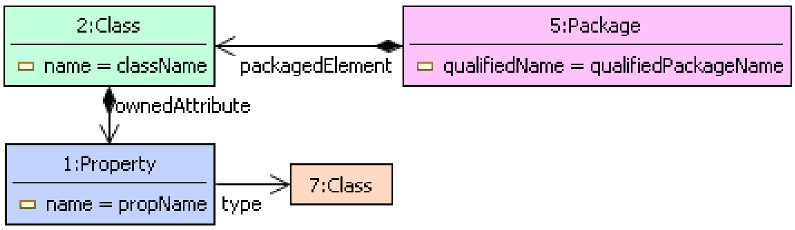
There are similarities between the structures defined in the MOF metamodelling language and in object-oriented programming languages. For the latter, refactoring pattern catalogues exist (such as [Fowler 1999]), which might usefully be applied to modelling languages. [Moha *et al.* 2009] provides a notation 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 (such as classes and attributes), [Moha *et al.* 2009] show that refactorings can be shared among them, but only consider 3 of XX 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 2.2.2). Defining a domain-specific language is one way in which abstraction can be realised for MDE (Section 2.3.1). 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.

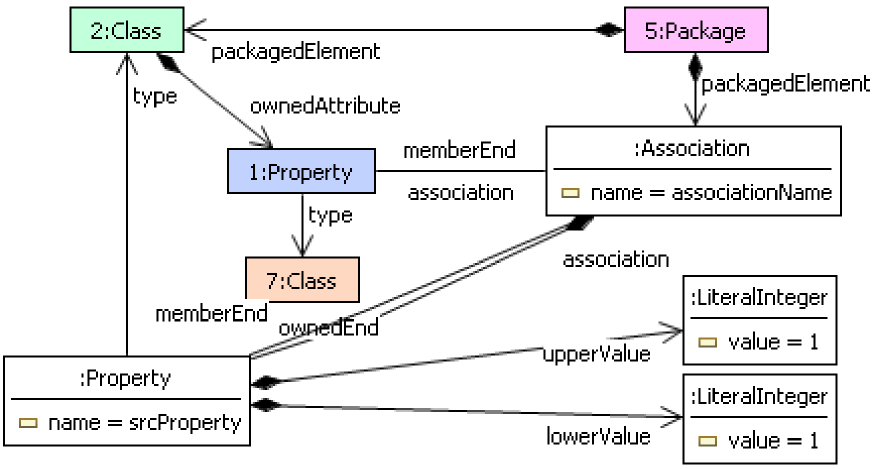
Eclipse, an extensible development environment, provides a library for building development tools for textual languages, LTK (language toolkit). LTK allows developers to specify – in Java – refactorings for their language, which can be invoked via the language editor. LTK makes no assumptions on the 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.* 2007] 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 5.3.5). Consequently, EWL, unlike LTK, abstracts over modelling frameworks.

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



(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].

[Kolovos *et al.* 2007] 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 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.

TODO: summary.

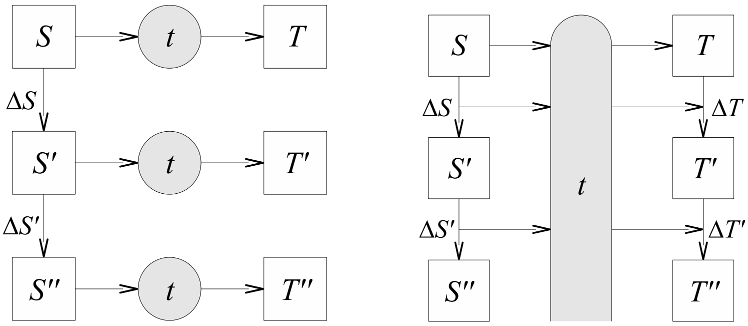
Model Synchronisation

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. Theoretical aspects of the traceability literature were reviewed in Section 3.1.4. This section analyses the application of traceability theory to model-driven engineering, focusing on the way in which traceability facilitates *model synchronisation*.

There is an abundance of model synchronisation approaches that extend or instrument existing model-to-model transformation languages. Declarative transformation languages lend themselves to the specification of bi-directional 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].

Primarily, incremental transformation has been used to address the scalability of model transformations, and this focus might be unhelpful. For large models, transformation execution time has been shown to be significantly reduced by using incremental transformation [Hearnden *et al.* 2006]. However,



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

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

[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 should seek to improve 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 more 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.

Some existing work provides a foundation for automating model synchronisation activities. Enriching a model-to-model transformation with traceability is discussed in [Jouault 2005], which contributes a generic higher-order transformation for this purpose. 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. Furthermore, 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].

While most of the model synchronisation literature focuses on synchronising models with other models, some papers consider synchronisation of models and text, usually in the context of code generation. The model-to-text language, Epsilon Generation Language (EGL) [Rose *et al.* 2008], produces traceability links between code generation templates and generated files. Sections of code can be marked protected, and are not overwritten by subsequent invocations of the code generation template. As described in [**?**], the MOFScript model-to-text language also provides protected sections and, in addition, stores traceability links in a structured manner. The traceability links described in [**?**] 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).

Model-metamodel Co-Evolution

When a metamodel evolves, any instance models may require migration to remain conformant. This activity is termed *model-metamodel co-evolution* and is subsequently referred to as *co-evolution*. Existing approaches to co-evolution are now explored.

[Sprinkle & Karsai 2004] were the first to describe co-evolution as distinct from the more general activity of model-to-model transformation. Sprinkle and Karsai identified the need for approaches that consider the specific requirements of co-evolution. In particular, the phrase "evolution, not revolution" was coined in [Sprinkle 2003] to highlight that, during co-evolution, the difference between source and target metamodels is often small.

Another key contribution was the way in which [Sprinkle & Karsai 2004] identified migration strategies as either *syntactic* or *semantic*. In the former, no information is known about the intention of the developer performing the evolution. In the latter, a lack of detailed information about the semantics of evolution can reduce the extent to which migration activities 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 automated migration:

- 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 to fully automate migration. Furthermore, tools that focus only upon syntax cannot perform migration 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 be preserved during migration: this is often the case in corrective evolution. However, it is crucial that the semantics are sound (i.e. conform to their specification) after migration. Addressing this kind of complexity is still an open research topic, to which this thesis contributes.

The approach to co-evolution outlined in [Sprinkle & Karsai 2004] requires that migration activities are specified by the developer. Instead, [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). The primary contribution made in [Sprinkle & Karsai 2004] was the definition of a process for performing co-evolution, shown in Figure 3.4. One key innovation is the way in which metamodel changes are classified. Gruschko et al. recognise that co-evolution in response to some types of metamodel change can be automated only when guided by a developer.

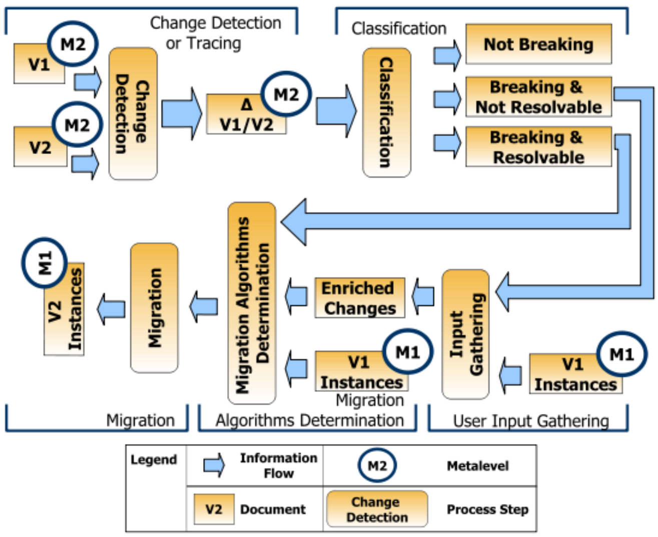


Figure 3.4: Envisaged approach to co-evolution, taken from [Gruschko *et al.* 2007].

[Wachsmuth 2007] classifies metamodel updates, and provides formal definitions of their impact on instance models. Wachsmuth was the first to employ higher-order model transformation[3] to generate a model-to-model transformation for performing co-evolution. However, the co-evolution strategies produced by [Wachsmuth 2007] are not automatically inferred. The most recent work on co-evolution includes a mechanism for automatically inferring co-evolution strategies [Cicchetti *et al.* 2008].

Automated migration is still an open research challenge. The most promising approaches (described in [Wachsmuth 2007, Cicchetti *et al.* 2008]) are still in their infancy, and key problems still need to be addressed. For example,

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

as discussed in Section 3.2.2, [Lerner 2000] highlights that analysis of a difference model can yield more than one feasible set of co-evolution strategies. The approaches discussed in [Wachsmuth 2007, Cicchetti *et al.* 2008] do not acknowledge this challenge.

Another open challenge is in identifying an appropriate notation for describing co-evolution. [Wachsmuth 2007, Cicchetti *et al.* 2008] use higher-order transformations. Although this is a sensible approach, co-evolution specialises model-to-model transformation: co-evolution only occurs in response to metamodel evolution. Therefore, devising a domain-specific language for specifying co-evolution strategies may facilitate increased expressiveness.

Until automated co-evolution is better understood and tools to support it become stable, improving existing technology to better support evolution is necessary. For example, the Eclipse Modelling Framework [Eclipse 2008a] cannot load models that no longer conform to their metamodel. This poses a problem for performing co-evolution manually. The author is not aware of any work that seeks to improve existing tooling to better facilitate manual migration.

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 3.5 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 co-evolution.

Furthermore, as co-evolution is often implemented as a specialisation of model-to-model transformation, Pilgrim et al.'s editor could be extended to permit visualisation of co-evolution for models with a visual concrete syntax. In this case, each plane would represent an instance conforming to different versions of the same metamodel.

3.3 Summary

Domain-specific languages underpin the implementation of almost all of the approaches discussed in this chapter. (Lever [Pizka & Jürgens 2007], for example, defines three domain-specific languages for evolving grammars and words, and for co-evolving grammars with programs). Yet the literature does not assess the efficacy of the DSLs, in particular for capturing the patterns common to evolution in their respective domains (databases, XML, grammarware, MDD environments). This is an area of research to which this thesis contributes, particularly in Chapter 5.

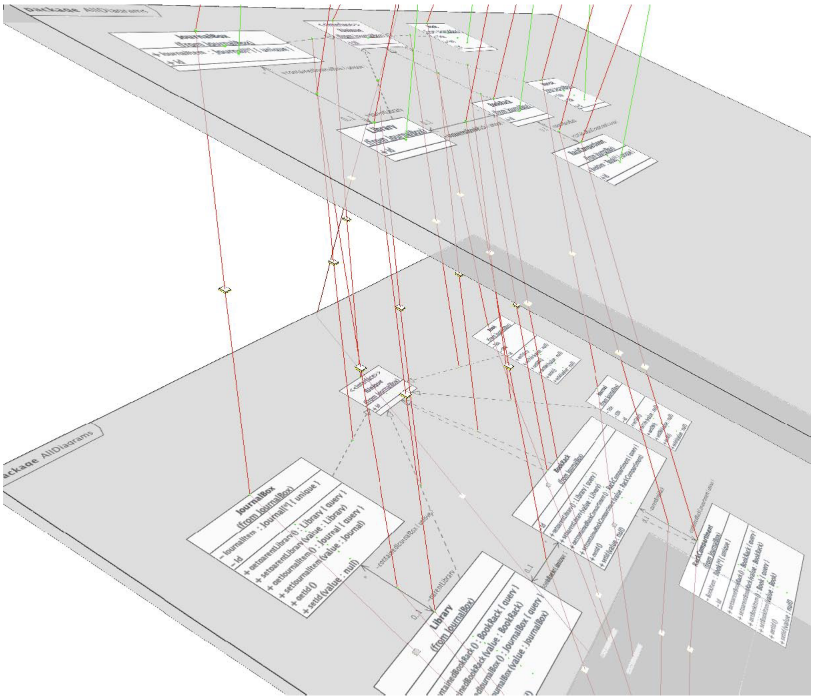


Figure 3.5: Visualising a transformation chain [Pilgrim *et al.* 2008].

Most existing approaches to managing software evolution are defined in work that uses a similar method: first, identify and categorise all feasible evolutionary changes. Next, design a taxonomy of operators that capture these changes (or a matching algorithm that detects the application of the changes). Finally, implement a tool that allows the developer to apply the evolutionary operators (or invoke the matching algorithm), and evaluate the tool on existing projects.

Most notably, Digg's work on program refactoring ([Dig & Johnson 2006b, Dig & Johnson 2006a, Dig *et al.* 2006, Dig *et al.* 2007]) follows a different method in which the first step analyses existing projects to identify common and feasible evolutionary changes. Identifying changes from existing projects has several benefits compared to the method typically used in managing software evolution, described above. Firstly, further research requirements can be identified from the solutions currently employed by existing projects. Secondly, related work can be more rigorously analysed and compared via application to existing projects. On the other hand, evaluating work produced by identifying changes from existing projects presents a challenge: more data may be required overall, as data used in the analysis should not be used in the evaluation.

As discussed in Section 1.4, this thesis follows a research method similar to that of Digg. Existing approaches to co-evolution and model synchronisation are compared using data taken from MDE projects that exhibit some degree of evolutionary change. From this analysis, research requirements are derived, and structures and process for managing evolution are implemented, evaluated, and then related back to the literature discussed in this chapter.

Bibliography

[37-Signals 2008] 37-Signals. Ruby on Rails [online]. [Accessed 30 June 2008] Available at: `http://www.rubyonrails.org/`, 2008.

[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.

[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.

[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.

[Bosch 1998] Jan Bosch. Design patterns as language constructs. *Journal of Object Oriented Programming*, 11(2):18–32, 1998.

[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.

[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.

[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.* 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.

[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. 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.

[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.

[Fritzsche *et al.* 2008] M. Fritzsche, J. Johannes, S. Zschaler, A. Zherebtsov, and A. Terekhov. Application of tracing techniques in Model-Driven Performance Engineering. In *Proc. ECMDA Traceability Workshop (ECMDA-TW)*, pages 111–120, 2008.

[Fuhrer *et al.* 2007] Robert M. Fuhrer, Adam Kiezun, and Markus Keller. Refactoring in the Eclipse JDT: Past, present, and future. In *Proc. Workshop on Refactoring Tools*, 2007.

[Gamma *et al.* 1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley, 1995.

[Garcés *et al.* 2009] Kelly Garcés, Frédéric Jouault, Pierre Cointe, and Jean Bézivin. Managing model adaptation by precise detection of metamodel changes. In *Proc. ECMDA-FA*, volume 5562 of *LNCS*, pages 34–49. Springer, 2009.

[Gosling *et al.* 2005] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java^{TM} Language Specification.* Addison-Wesley, Boston, MA, USA, 2005.

[Gronback 2006] Richard Gronback. Introduction to the Eclipse Graphical Modeling Framework. In *Proc. EclipseCon*, Santa Clara, California, 2006.

[Gruschko *et al.* 2007] Boris Gruschko, Dimitrios S. Kolovos, and Richard F. Paige. Towards synchronizing models with evolving metamodels. In *Proc. Workshop on Model-Driven Software Evolution*, 2007.

[Guerrini *et al.* 2005] Giovanna Guerrini, Marco Mesiti, and Daniele Rossi. Impact of XML schema evolution on valid documents. In *Proc. Workshop on Web Information and Data Management*, pages 39–44, 2005.

[Hearnden *et al.* 2006] David Hearnden, Michael Lawley, and Kerry Raymond. Incremental model transformation for the evolution of model-driven systems. In *Model Driven Engineering Languages and Systems*, volume 4199 of *LNCS*, pages 321–335. Springer, 2006.

[Herrmannsdoerfer *et al.* 2009] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Juergens. COPE - automating coupled evolution of metamodels and models. In *Proc. ECOOP*, volume 5653 of *LNCS*, pages 52–76. Springer, 2009.

[Hussey & Paternostro 2006] Kenn Hussey and Marcelo Paternostro. Advanced features of EMF. Tutorial at EclipseCon 2006, California, USA. [Accessed 07 September 2009] Available at: `http://www.eclipsecon.org/2006/Sub.do?id=171`, 2006.

[IBM 2005] IBM. Emfatic Language for EMF Development [online]. [Accessed 30 June 2008] Available at: `http://www.alphaworks.ibm.com/tech/emfatic`, 2005.

[IRISA 2007] IRISA. Sintaks. `http://www.kermeta.org/sintaks/`, 2007.

[Jouault & Kurtev 2005] Frédéric Jouault and Ivan Kurtev. Transforming models with ATL. In *Proc. Satellite Events at the International Conference on Model Driven Engineering Languages and Systems*, volume 3844 of *LNCS*, pages 128–138. Springer, 2005.

[Jouault 2005] Frédéric Jouault. Loosely coupled traceability for ATL. In *Proc. ECMDA-FA Workshop on Traceability*, 2005.

[Kataoka *et al.* 2001] Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David Notkin. Automated support for program refactoring using invariants. In *Proc. International Conference on Software Maintenance*, pages 736–743. IEEE Computer Society, 2001.

[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.* 2006] 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.* 2007] 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.* 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.

[Lago *et al.* 2009] Patricia Lago, Henry Muccini, and Hans van Vliet. A scoped approach to traceability management. *Systems and Software*, 82(1):168–182, 2009.

[Lämmel & Verhoef 2001] R. Lämmel and C. Verhoef. Semi-automatic grammar recovery. *Software - Practice and Experience*, 31(15):1395–1438, 2001.

[Lämmel 2001] R. Lämmel. Grammar adaptation. In *Proc. Formal Methods for Increasing Software Productivity (FME), International Symposium of Formal Methods Europe*, volume 2021 of *LNCS*, pages 550–570. Springer, 2001.

[Lämmel 2002] R. Lämmel. Towards generic refactoring. In *Proc. ACM SIGPLAN Workshop on Rule-Based Programming*, pages 15–28. ACM, 2002.

[Lehman 1969] Meir M. Lehman. The programming process. Technical report, IBM Res. Rep. RC 2722, 1969.

[Lehman 1978] Meir M. Lehman. Programs, cities, students - limits to growth? *Programming Methodology*, pages 42–62, 1978.

[Lehman 1980] Meir M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213–221, 1980.

[Lehman 1985] Meir M. Lehman. *Program evolution: processes of software change*. Academic, 1985.

[Lehman 1996] Meir M. Lehman. Laws of software evolution revisited. In *Proc. European Workshop on Software Process Technology*, pages 108–124, 1996.

[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.

[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.

[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 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. 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. 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.

[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.

[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.

[Pool 1997] R. Pool. *Beyond Engineering: How Society Shapes Technology.* Oxford University Press, 1997.

[Porres 2003] Ivan Porres. Model refactorings as rule-based update transformations. In *Proc. UML*, volume 2863 of *LNCS*, pages 159–174. Springer, 2003.

[Ráth *et al.* 2008] István Ráth, Gábor Bergmann, András Ökrös, and Dániel Varró. Live model transformations driven by incremental pattern matching. In *Proc. ICMT*, volume 5063 of *LNCS*, pages 107–121. Springer, 2008.

[Rising 2001] Linda Rising, editor. *Design patterns in communications software.* Cambridge University Press, 2001.

[Rose *et al.* 2008] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona A.C. Polack. The Epsilon Generation Language. In *Proc. European Conference on Model Driven Architecture – Foundations and Applications*, volume 5095 of *LNCS*, pages 1–16. Springer, 2008.

[Rose *et al.* 2009a] Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Enhanced automation for managing model and metamodel inconsistency. In *Proc. ASE.* 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, 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*, 2010.

[Rose *et al.* 2010b] Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Model migration with flock. In *In preparation*, 2010.

[Selic 2003] Bran Selic. The pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003.

[Sjøberg 1993] Dag I.K. Sjøberg. Quantifying schema evolution. *Information & Software Technology*, 35(1):35–44, 1993.

[Sommerville 2006] Ian Sommerville. *Software Engineering*. Addison-Wesley Longman, 2006.

[Sprinkle & Karsai 2004] Jonathan Sprinkle and Gábor Karsai. A domain-specific visual language for domain model evolution. *Journal of Visual Languages and Computing*, 15(3-4):291–307, 2004.

[Sprinkle 2003] Jonathan Sprinkle. *Metamodel Driven Model Migration*. PhD thesis, Vanderbilt University, TN, USA, 2003.

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

[Steinberg *et al.* 2008] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2008.

[Su *et al.* 2001] Hong Su, Diane Kramer, Li Chen, Kajal T. Claypool, and Elke A. Rundensteiner. XEM: Managing the evolution of XML documents. In *Proc. Workshop on Research Issues in Data Engineering*, pages 103–110, 2001.

[Tratt 2008] Laurence Tratt. A change propagating model transformation language. *Journal of Object Technology*, 7(3):107–124, 2008.

[Varró & Balogh 2007] Dániel Varró and András Balogh. The model transformation language of the VIATRA2 framework. *Science of Computer Programming*, 68(3):187–207, 2007.

[Vries & Roddick 2004] Denise de Vries and John F. Roddick. Facilitating database attribute domain evolution using meso-data. In *Proc. Workshop on Evolution and Change in Data Management*, pages 429–440, 2004.

[W3C 2007a] W3C. W3C XML Schema 1.1 Specification [online]. [Accessed 30 June 2008] Available at: `http://www.w3.org/XML/Schema`, 2007.

[W3C 2007b] W3C. World Wide Web Consortium [online]. [Accessed 30 June 2008] Available at: `http://www.w3.org/`, 2007.

[Wachsmuth 2007] Guido Wachsmuth. Metamodel adaptation and model co-adaptation. In *Proc. ECOOP*, volume 4609 of *LNCS*, pages 600–624. Springer, 2007.

[Wallace 2005] Malcolm Wallace. Modular architectural representation and analysis of fault propagation and transformation. *Electronic Notes in Theoretical Computer Science*, 141(3):53–71, 2005.

[Watson 2008] Andrew Watson. A brief history of MDA. *Upgrade*, 9(2), 2008.

[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.