Evolution in Model-Driven Engineering

Louis M. Rose

June 14, 2010

Contents

Chapter 2

Background

Before reviewing software evolution research, it is first necessary to survey literature from model-driven engineering, which is the engineering approach in which the thesis research is conducted. Section 2.1 introduces the terminology and fundamental principles used in model-driven engineering. Section 2.2 reviews guidance and three methods for performing model-driven engineering. Section 2.3 describes contemporary model-driven engineering environments. The related areas of domain-specific languages, language-oriented programming and grammarware are discussed in Section 2.4. Finally, the benefits of and current challenges for model-driven engineering are described in Section 2.5.

2.1 MDE Terminology and Principles

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

2.1.1 Models

When used here, the term *model* has the same meaning as given in [Kolovos *et al.* 2006c]: a model is a description of a phenomenon of interest, and may have either a textual or graphical representation. A model provides an abstraction over a real-world object, which enables engineers of differing disciplines to reason about that object.

Abstraction is the primary reason for and the primary goal of modelling. [Evans 2004] proposes the use of models throughout the development process to capture and communicate domain knowledge and to shape the structure of the resulting software. Evans emphasises the importance of modelling and a process, which he terms *refactoring to deeper insight*, that seeks incremental improvements to models.

> Distillation is the process of separating the components of a mixture to extract the essence in a form that makes it more valuable and useful. A model is a distillation of knowledge. With every refactoring to deeper insight, we abstract some crucial aspect of domain knowledge and priorities. [Evans 2004, pg397]

[Martin & Martin 2006, ch14] notes that, in some engineering disciplines, models are used to reduce risk. Structural engineers build models of bridges. Aerospace engineers build models of aircraft. In these disciplines, a model is used to determine the efficacy of the real thing and, moreover, is cheaper to build and test than the real thing, often by a huge factor. The produce of many engineering disciplines is physical and the manufacturing process costly. Often, software models are not cheaper by a huge factor to build and test than the software they represent. Consequently, [Martin & Martin 2006, ch14] prescribes software modelling for communicating and reasoning about a design, and not as a long-term replacement for real, working software.

All software has a *domain*, the activities or business of its users. The domain of a library's lending system includes books, people and loans. [Evans 2004] prescribes principles and practices for building software in a way that emphasises the underlying domain, while tackling its complexity. In [Evans 2004], domain models are key – they are used to shape the solution's design, to define a common vocabulary for communication between team members, and to distinguish interesting and uninteresting elements of the domain. According to [Evans 2004], domain models are key to software development.

2.1.2 Metamodelling

In model-driven engineering, models are structured (conform to a well-defined set of syntactic and semantic constraints) rather than unstructured [Kolovos 2009]. A *modelling language* is the set of syntactic and semantic constraints used to define the structure of group of related models. In model-driven engineering, a modelling language is often specified as a model and, hence the term *metamodel* is often used in place of *modelling language*.

Metamodels facilitate model interchange and, hence, interoperability between modelling tools. For this reason, Evans recommends that software engineers "use a well-documented shared language that can express the necessary domain information as a common medium of communication." [Evans 2004, pg377]. To support this recommendation, Evans discusses Chemical Markup Language (CML), a standardised language, which has facilitated the interoperability of tools (such as JUMBO Browser, which creates graphical views of chemical structures) developed by various institutions.

A metamodel typically comprises three categories of constraint:

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

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

- **The semantics** identifies the meaning of the modelling concepts in the particular domain of language. For example, consider a modelling language defined to describe genealogy, and another to describe flora. Although both languages may define a tree construct, the semantics of a tree in one is likely to be different from the semantics of a tree in the other. The semantics of a modelling language may be specified rigorously, by defining a reference semantics in a formal language such as Z [ISO/IEC 2002], or in a semi-formal manner by employing natural language.

Concrete syntax, abstract syntax and semantics are used together to specify modelling languages. There are many other ways of defining languages, but this approach (first formalised in [Álvarez *et al.* 2001]) is common in model-driven engineering: a metamodel is often used to define abstract syntax, a grammar or text-to-model transformation to specify concrete syntax, and code generators, annotated grammars or behavioural models to effect semantics.

MOF

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

Because MOF is a modelling language for describing modelling languages, it is sometimes termed a metamodelling language. A simplified fragment of the UML defined in MOF, is shown in Figure 5.3. The concrete syntax of MOF borrows is similar to the concrete syntax of UML class diagrams:

- Modelling constructs are drawn as boxes. The name of each modelling construct is emboldened. The name of abstract (uninstantiable) constructs are italicised.

- Attributes are contained within the box of their modelling construct. Each attribute has a name, a type (prefixed with a colon) and may define a default value (prefixed with an equals sign).

- Generalisation is represented using a line with an open arrow-head.

- References are specified using a line. An arrow illustrates the direction in which the reference may be traversed (no arrow indicates bidirectionality). Labels are used to name and define the multiplicity of references.

- Containment references are specified by including a solid diamond on the containing end.

mof.png

Figure 2.1: A fragment of the UML metamodel defined in MOF, from [OMG 2007a].

In the past, the means for describing modelling constructs has been inconsistent between modelling languages. For example, both entity-relationship (ER) diagrams and UML class diagrams can be used to specify models of structured data but, as [Frankel 2002, pg97] notes, similar constructs from ER diagrams and UML class diagrams have different concrete syntax. MOF seeks to standardise the way in which modelling languages are defined.

2.1.3 Model-Driven Engineering

Model-driven engineering (MDE) is a principled approach to software engineering in which models are produced throughout the engineering process. Models are manipulated throughout development to produce software. This thesis uses the term *model management*, defined in [Kolovos 2009], to refer to development activities that manipulate models for the purpose of producing software. Typical model management activities are discussed in this section.

Model Transformation

Model transformation is a development activity in which software artefacts are derived from others, according to some well-defined specification. Model transformations are specified between modelling languages (model-to-model transformation), between modelling languages and textual artefacts (model-to-text-transformation) and between textual artefacts and modelling languages (text-to-model transformation).

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

Transformations specified between the same source and target metamodel are termed *endogenous*, while transformation specified between different source and target metamodels are term *exogenous*. Endogenous transformations can typically be specified with a more compact syntax than exogenous transformations, because there is no need to specify mappings between source and target types.

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

Endogenous, existing-target transformations are used to perform small or incremental updates to models, such as refactorings, which will be discussed in Chapter 3.

Model-to-Model (M2M) Transformation M2M transformation has been characterised as the heart-and-soul of MDE [?]. Large and complex systems can be represented using several interdependent models. By automating the derivation of models from others, model transformation has the potential to reduce the cost of engineering large and complex systems.

[Kolovos *et al.* 2008a] notes that the current consensus is that hybrid languages, such as QVT [OMG 2005] and ATL [Jouault & Kurtev 2005] are more suitable for specifying model transformation than pure imperative or declarative languages.

Model-to-Text (M2T) Transformation M2T transformation is an important model management task with a number of applications, including model serialisation (enabling model interchange); code and documentation generation; and model visualisation and exploration. In 2005, the OMG [OMG 2008c] recognised the lack of a standardised in M2T transformation with its M2T Language Request for Proposals [?]. In response, various M2T languages have been developed, including JET [?], XPand and MOFScript.

Because M2T transformation is used to produce unstructured artefacts, M2T transformation has different requirements to M2M transformation. For instance, code generators often provide mechanisms for specifying sections of code to be completed manually and for preserving those hand-written sections. Traceability between structured and unstructured artefacts is also a key requirement for model-to-text (and text-to-model) transformation, and is discussed further in Chapter 3.

Text-to-Model (T2M) Transformation T2M transformation is most often implemented as a parser that produces a model. Parser generators such as ANTLR [?] can be used to produce a structured artefact (such as an abstract syntax tree) from text. T2M tools are built atop parser generators and post-process the structured artefacts such that they conform to the target metamodel. Xtext [?] and EMFtext [?] are contemporary examples of T2M tools that, given a grammar and a target metamodel, will automatically generate a parser that transforms text to a model.

Model Validation

Model validation provides a mechanism for managing the integrity of the software developed using MDE. A model that omits information is said to be *incomplete*, while related models that suggest differences in the underlying phenomena are said to be *contradicting*. Incompleteness and contradiction are two examples of *inconsistency*. In MDE, inconsistency is detrimental, because, when artefacts are automatically derived from each other, the inconsistency of one artefact might be propagated to others. Model validation is used to detect, report and reconcile inconsistency throughout a MDE process.

[Kolovos 2009] observes that inconsistency detection is inherently pattern-based and, hence, higher-order languages are more suitable for model validation than 3G languages (such as Java). The Object Constraint Language (OCL) [OMG 2006], an OMG standard, can be used to specify consistency constraints on UML and MOF models. Unlike OCL, the xlinkit toolkit [Nentwich *et al.* 2003] can be used for specifying inter-model consistency constraints.

Further model management activities

In addition to model transformation and validation, further examples of model management activities include model comparison (e.g. [Kolovos *et al.* 2006b]), in which a *trace* of similar and different elements is produced from two or more models, and model merging or weaving (e.g. [**?**]), in which a two or more models are combined to produce a unified model.

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

2.1.4 Summary

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

2.2 MDE Methods

For performing model-driven engineering, new practices and processes have been proposed. Proponents of MDE have produced guidance and methods for model-driven engineering. This section discusses the guidance for MDE set out in the Model-Driven Architecture [OMG 2008b] and the methods described by [Stahl *et al.* 2006, Kelly & Tolvanen 2008, Greenfield *et al.* 2004].

2.2.1 Model-Driven Architecture (MDA)

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

A PIM provides an abstract, implementation-agnostic view of the solution. Successive PSMs provide increasingly more implementation detail. Inter-model mappings are used to forward- and reverse-engineer these models, as depicted in Figure 2.2.

Figure 2.2: Interactions between a PIM and several PSMs.

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

McNeile [McNeile 2003] identifies two ways in which engineers are utilising MDA. Both interpretations begin with a PIM and vary in the way they are used to produce executable code:

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

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

Translationists must encode behaviour in their PIMs [Mellor & Balcer 2002], whereas elaborationists have a choice, frequently electing to specify behaviour in PSMs or in code [Kleppe *et al.* 2003].

The MDA prescribes a set of standards for MDE. The MDA allocates standards to one of four tiers, representing different levels of model abstraction. Members of each tier are instances of the members of parent tiers. These tiers can be seen in Figure 2.3, and a short discussion based on [Kleppe *et al.* 2003, Section 8.2] follows.

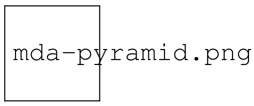

Figure 2.3: The tiers of standards used as part of MDA.

The base of the pyramid, tier M0, describes the real-world. When modelling a business, this tier is used to describe items of the business itself, such as a real customer or an invoice. When modelling software, M0 instances describe the software representation of such items. M1 contains a model of the concepts in M0, for example a customer may be represented as a class with attributes. The M2 tier describes the model of the modelling language used to describe elements of M1. For example, if UML [OMG 2007a] were used to describe concepts as classes in the M1 tier, M2 would contain the UML metamodel. Finally, M3 is the meta-metamodel layer, which provides a description of the metamodel used in M2. M3 is necessary to permit reasoning about metamodels (such as the UML), and to enable tool standardisation. The OMG defines the Meta-Object Facility (MOF) [OMG 2008a] as the sole inhabitant of the M3 tier.

2.2.2 Methods for MDE

Several methods to MDE are prevalent today. In this section, three of the most established are discussed: Architecture-Centric Model-Driven Software Development [Stahl *et al.* 2006], Domain-Specific Modelling [Kelly & Tolvanen 2008] and Microsoft's Software Factories [Greenfield *et al.* 2004]. All three methods have been defined from a pragmatic standpoint (i.e. they have been used repeatedly to solve problems in industry). The methods vary in the extent to which they follow the guidelines set out by MDA.

Architecture-Centric Model-Driven Software Development

Model-Driven Software Development is the term given to MDE by in [Stahl *et al.* 2006]. The style of MDE that Stahl et al. describe, *architecture-centric model-driven software development* (AC-MDSD), focuses on generating the infrastructure of large-scale applications. For example, a typical J2EE application contains concepts (such as EJBs, descriptors, home and remote interfaces) that "admittedly contain domain-related information such as method signatures, but which also exhibit a high degree of redundancy" [Stahl *et al.* 2006]. It is this redundancy that AC-MDSD seeks to remove by using code generators, requiring only the domain-related information to be specified.

AC-MDSD applies more of the MDA guidelines than the other methods discussed below. For instance, AC-MDSD supports the use of a general-purpose modelling language for specifying models. [Stahl *et al.* 2006] utilise UML in many of their examples, which demonstrate how AC-MDSD may be used to enhance the productivity, efficiency and understandability of software development. In these examples, models are annotated using UML profiles to describe domain-specific concepts.

Domain-Specific Modelling

[Kelly & Tolvanen 2008] present a method for MDE termed Domain-Specific Modelling (DSM). DSM is based on the translationist interpretation of MDA; DSM seeks to translate models containing concepts from the problem domain to full code. In motivating the need for DSM, Kelly and Tolvanen state that large productivity gains were made when third-generation programming languages were used in place of assembler, and that no paradigm shift has since been able to replicate this degree of improvement. Tolvanen[1] notes that DSM focuses on increasing the productivity of software engineering by allowing developers to specify solutions by using models that describe the application domain.

To perform DSM, expert developers define:

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

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

As the development of these three artefacts requires significant effort from expert developers, Tolvanen[1] states that DSM should only be applied if more than three problems specific to the same domain are to be solved.

Tools for defining domain-specific modelling languages, editors and code generators enable DSM [Kelly & Tolvanen 2008]. Reducing the effort required to specify these artefacts is key to the success of DSM. In this respect, DSM resembles a programming paradigm popular in the *domain-specific language* (DSL) community, termed *language-oriented programming* (LOP), which also requires tools to simplify the specification of new languages. DSLs and LOP are discussed further in Section 2.4.

Throughout [Kelly & Tolvanen 2008], examples from industrial partners are used to illustrate that DSM can greatly improve developer productivity. Unlike MDA, DSM seems to be optimised for increasing productivity, and less concerned with portability or maintainability. Therefore, DSM is less suitable for engineering applications that frequently interoperate with – and are underpinned by – changing technologies.

Microsoft Software Factories

Greenfield [Greenfield *et al.* 2004, pg159] states that industrialisation of the automobile industry has addressed problems with economies of scale (mass production) and scope (product variation). Software Factories, a software engineering method developed at Microsoft, seek to address problems with economies of scope in software engineering by borrowing concepts from product-line engineering. Greenfield [Greenfield *et al.* 2004] argues that, unlike many other engineering disciplines, software development requires considerably more development effort than production effort in that scaling software development to account for scope is significantly more complicated then mass production of the same software system.

The Software Factories method [Greenfield *et al.* 2004] prescribes a bottom-up approach to abstraction and re-use. Development begins by producing prototypical applications. The common elements of these applications are identified and abstracted into a product-line. When instantiating a product, models are used to specify product variance (e.g. by selecting particular product features). To generate these models, tools for use with Software Factories provide mechanisms for defining wizards and feature-based configuration selection dialogues. By contrast, DSM relies upon the use of concrete syntax for producing models that describe product variance. By providing explanations that assist in making decisions, the wizards used in Software Factories guide users towards best practices. Greenfield et al. state that "moving from totally-open ended hand-coding to more constrained forms of specification [such as wizard-based feature selection] are the key to accelerating software development" [Greenfield *et al.* 2004, pg179].

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

Finally, Microsoft prescribes the use of domain-specific languages (discussed in Section **??**) for describing models in conjunction with Software Factories, rather than a general-purpose modelling language, as Microsoft believes that the latter often have imprecise semantics [Greenfield *et al.* 2004].

2.2.3 Summary

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

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

Each method has a different focus. AC-MDSD seeks to reduce the amount of boilerplate code being generated, particularly in enterprise applications. Software Factories concentrate on providing different viewpoints into the system, allowing different domain experts to collaborate when specifying a system. DSM aims to decrease the time taken to develop software solutions to instances of the problem domain.

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

2.3 MDE Tools

For model-driven engineering to be applicable in the large, and to complex systems, mature and powerful tools and languages must be available. Such tools and languages are beginning to emerge, and this section discusses the state of current MDE development environments, which are typically a combination of tools and languages.

This section provides a brief overview of the Eclipse Modelling Framework [Eclipse 2008a], which underpins many of MDE tools and languages, facilitating their interoperability. Subsequently, a discussion of Epsilon [Eclipse 2008c], an extensible platform for the specification of model management languages, is presented. The highly extensible nature of Epsilon (which is described below) makes it an ideal host for the rapid prototyping of languages and exploring research hypotheses.

2.3.1 Eclipse Modelling Framework

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

EMF provide metamodel-specific editors for loading, storing and constructing models. EMF model editors comprise a navigation view that depicts the model as a structure and a properties view that is used to specify the values of model element features. By default, EMF editors represent models on disk as XMI 2.1 [**?**] documents. Figure 2.4 shows an EMF model editor for a simplistic state machine language.

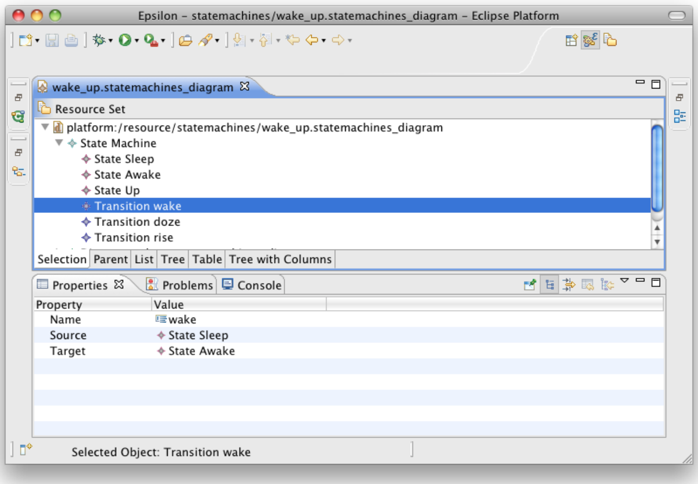

Figure 2.4: EMF state machine model editor.

Users of EMF can define their own metamodels in Ecore and generate a corresponding model editor. EMF provides both a tree-based editor (Figure 2.5) and a diagrammatic editor (Figure 2.6) for constructing metamodels. The latter uses syntax taken from UML class diagrams. An extension to EMF provides a textual editor for metamodels (Figure 2.7).

The range of concrete syntaxes for Ecore models presents a challenge for tools that wish to augment the way in which metamodels are defined (e.g. for

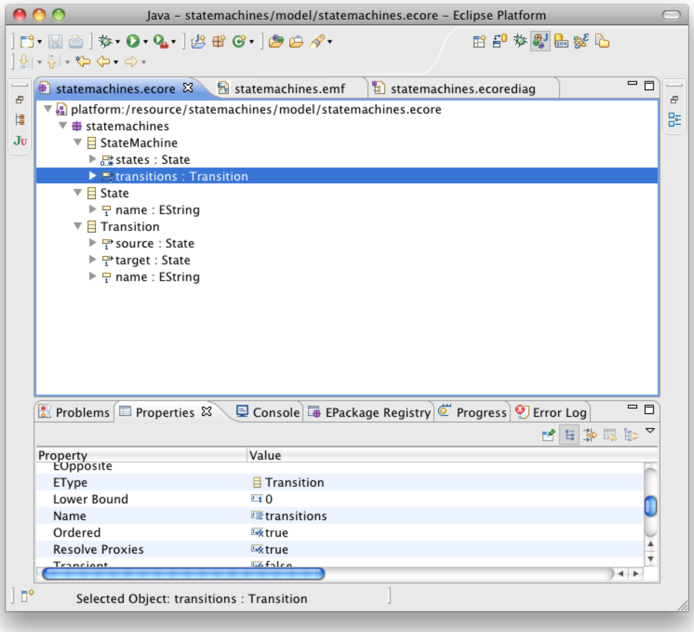

Figure 2.5: EMF tree-based metamodel editor.

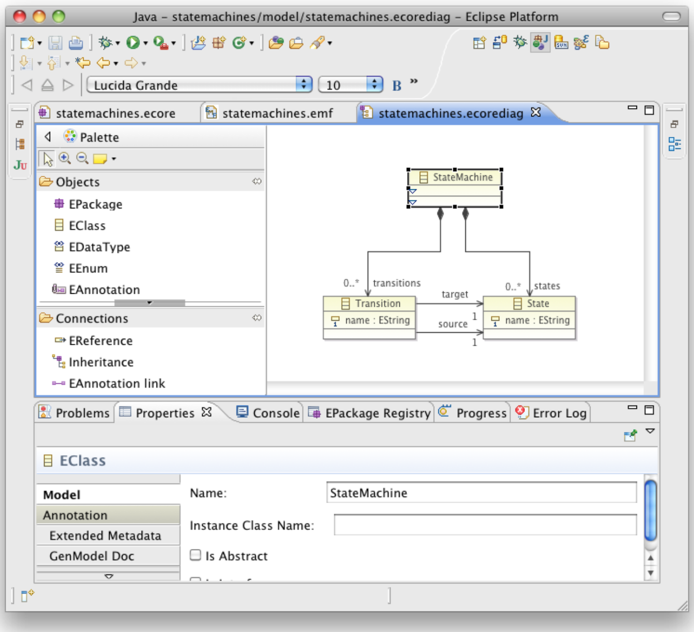

Figure 2.6: EMF diagrammatic metamodel editor.

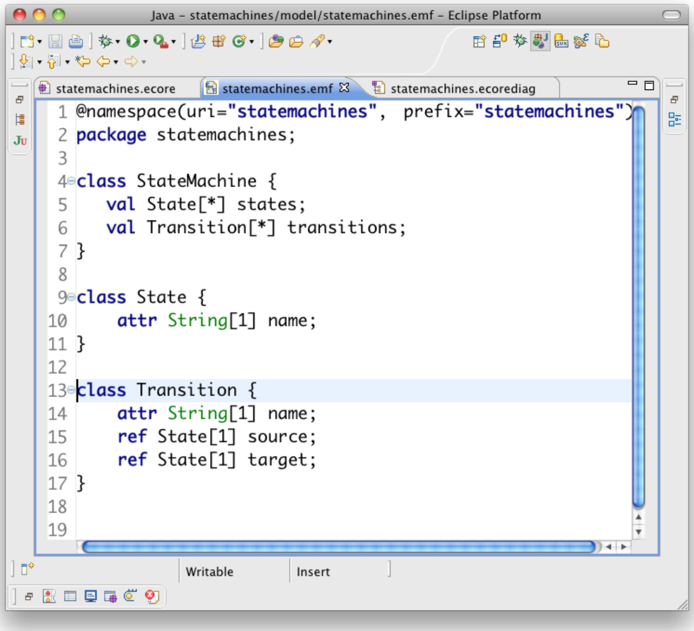

Figure 2.7: EMF textual metamodel editor.

specifying semantics). Extensions made to one type of metamodel editor (e.g. tree-based) will not be automatically be available in others (e.g. visual and textual). However, EMF does facilitate the programmatic monitoring of model changes, which can be used for implementing metamodel extensions, as discussed in Chapter 5.

From an Ecore model, EMF can generate a metamodel-specific model editor. The metamodel specified in Figure **??** was used to generate the Java code for the model editor shown in Figure 2.4. Because model editors are generated from metamodels, models and metamodels are kept separate in EMF. Consequently, metamodel changes cannot be propagated directly to models.

The Graphical Modeling Framework (GMF) [Gronback 2006] is used to specify graphical concrete syntax for metamodels defined in EMF. GMF itself uses a model-driven approach: users specify several models, which are combined, transformed and then used to generate code for the resulting graphical editor. Figure 2.8 shows a model editor produced with GMF for the simplistic state machine language described above.

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

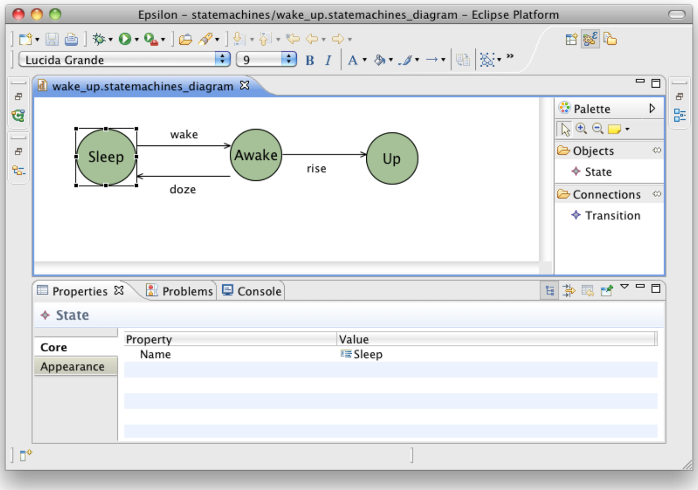

Figure 2.8: GMF state machine model editor.

2.3.2 Epsilon

The Extensible Platform for Specification of Integrated Languages for mOdel maNagement (Epsilon) [Eclipse 2008c] is a suite of tools and domain-specific languages for MDE. Epsilon comprises several integrated model management languages – built atop a common infrastructure – for performing tasks such as model merging, model transformation and inter-model consistency checking [Kolovos 2009].

Figure 2.9 illustrates the various components of Epsilon.

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

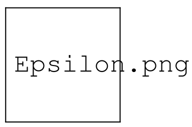

Figure 2.9: The architecture of Epsilon, taken from [Rose *et al.* 2008].

The architecture of Epsilon promotes reuse when building task-specific model management languages and tools. Each Epsilon language can be reused wholesale in the production of new languages. Ideally, the developer of a new language only has to design language concepts and logic that do not already exist in Epsilon languages. As such, new task-specific languages can be implemented in a minimalistic fashion. This claim has been demonstrated in [Rose *et al.* 2008], which describes the Epsilon Generation Language (EGL) for specifying model-to-text transformation.

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

As shown in Figure 2.9, every Epsilon language re-uses EOL, so improvements to this language enhance the entire platform. EOL also allows developers to delegate computationally intensive tasks to extension points, where the task can be authored in Java.

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

2.4 Research Relating to MDE

2.4.1 Domain-Specific Languages

For a set of closely-related problems, a specific, tailored approach is likely to provide better results than instantiating a generic approach for each problem [Deursen *et al.* 2000]. The set of problems for which the specific approach outperforms the generic approach is termed the problem domain. A *domain-specific programming language* (often called a *domain-specific language* (DSL)) enables solutions in a particular problem domain to be encoded.

Like modelling languages, DSLs describe abstract syntax. Furthermore, a common language can be used to define DSLs (e.g. EBNF [ISO/IEC 1996]), like the use of MOF for defining modelling languages. In addition to abstract syntax, DSLs always define a textual concrete syntax, whereas modelling languages can utilise a graphical concrete syntax (e.g. UML), a textual concrete syntax (e.g. Human-Usable Textual Notation (HUTN) [OMG 2004]) or no concrete syntax. Therefore, DSLs can be thought of as a subset of (domain-specific) modelling languages. As such, it is useful to identify the ways in which DSLs are being used for software development, in addition to the survey of approaches to performing MDE.

Cobol, Fortran and Lisp started life as DSLs for solving problems in the domains of business processing, numeric computation and symbolic processing respectively, and evolved to become general-purpose programming languages [Deursen *et al.* 2000]. DSLs are often designed to be very simple, especially at inception; they can grow to become complicated (e.g. SQL). However, a DSL cannot be used to program an entire application. Within their domain, simple DSLs are easy to read, understand and edit [Fowler 2005].

A typical approach to constructing a DSL (termed *embedding* a DSL) involves describing the domain using constructs from a general-purpose language (the *host*), such as classes, interfaces and message passing in an object-oriented language [Dmitriev 2004]. Examples of embedding a DSL include the frameworks for working with collections that are included in some programming languages (e.g. STL for C++, the Collections API for Java). Some languages are better hosts. For example, Fowler [Fowler 2005] proposes Ruby as a suitable host due to its "unintrusive syntax and flexible runtime evaluation." Graham [Graham 1993] describes a related style of development in Lisp, where macros are used to translate domain-specific concepts to Lisp abstractions.

However, [Dmitriev 2004] reports that embedding a DSL is often unsatisfactory, as the problem domain must be constructed by using the concepts specified in the host, which is a general-purpose language. This leads to a mismatch between domain and programming abstractions, which must be bridged by skilled developers. [Dmitriev 2004] suggests that a good DSL should not be bound to programming language abstractions. Developing a translation for programs written in a DSL to programs written in a general-purpose language is one alternative to embedding. Programs written in simple DSLs are often easy to translate to programs in an existing general-purpose language. Approaches to translation include preprocessing; building or generating an interpreter or compiler; or extending an existing compiler or interpreter [Dmitriev 2004].

DSLs have been successfully used as part of application development in many domains, as described in [Deursen *et al.* 2000]. They have been used in conjunction with general-purpose languages to build systems rapidly and to improve productivity in the development process (e.g. DSLs for the automation of system deployment and configuration). More recently, some developers are building complete applications by stitching together DSLs, in a style of development called Language-Oriented Programming.

2.4.2 Language-Oriented Programming

[Ward 1994] coins the term Language-Oriented Programming (LOP) to describe a style of development in which a very high-level language is used to encode the problem domain. Simultaneously, a compiler is developed to translate programs written in the high-level language to an existing programming language. Ward describes how this approach to programming can enhance the productivity of development and the understandability of a system. Additionally, Ward mentions the way in which multiple very high-level languages could be layered to separate domains.

The high-level languages that Ward discusses are domain-specific. [Fowler 2005] notes that combining DSLs to solve a problem is not a new technique. Traditionally, UNIX has encouraged developers to use small (domain-specific) languages (such as awk, make, sed, lex, yac) together to solve problems. Lisp (and, more recently, Ruby) programmers often construct domain-specific languages when developing programs [Graham 1993]. Smalltalk also has a strong tradition of this style of development [Fowler 2005].

To fully realise the benefits of LOP, the development effort required to construct DSLs must be minimised. Two approaches seem to be prevalent. The first advocates using a highly dynamic, reflexive and extensible programming language. Clark terms this category of language a *superlanguage* [Clark *et al.* 2008]. The superlanguage permits new DSLs to re-use constructs from existing DSLs, which simplifies development.

A *language workbench* [Fowler 2005] is an alternative means for simplifying DSL development. Language workbenches provide tools, wizards and DSLs for defining abstract and concrete syntax, for constructing editors and for specifying code generators.

For defining DSLs, the main difference between using a language workbench or a superlanguage is the way in which semantics of language concepts are encoded. In a language workbench, a typical approach is to write a generator for each DSL (e.g. MPS [JetBrains 2008]), whereas a superlanguage often requires that semantics be encoded in the definition of language constructs (e.g. XMF [Ceteva 2008]).

[Clark *et al.* 2008] acknowledges that a modern development environment for a superlanguage is an important concern. Therefore, the success of both LOP approaches depends, to some extent, upon the quality of their development environment (or workbench). Dependency on language workbenches is a key difference between LOP and MDE for two reasons:

1. The emphasis for LOP is in defining (textual) concrete syntaxes. Tools for MDE often provide an editor for manipulating abstract syntax directly, and constructing models using a concrete syntax is optional. Graphical (diagrammatic) concrete syntaxes are also popular when modelling.

2. MDE tools frequently support many types of model-management operation (such as model-to-model transformation and model merging), while language workbenches concentrate solely on translating DSL programs to code using generators. (Superlanguages do not need to provide any facilities for manipulating the abstract syntax directly).

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

In summary, LOP addresses many of the same issues with traditional development as MDE, but requires a different style of tool. LOP tools focus more on the integration of distinct DSLs, and providing editors and code generators for them; while MDE tools concentrate more on model management operations, such as model-to-model transformation.

Jetbrains MPS: http://www.jetbrains.com/mps/index.html http://blogs.jetbrains.com/mps/ http://confluence.jetbrains.net/display/MPS/MPS+Userhttp://www.jetbrains.net/devnet/community/mps

2.4.3 Grammarware

- Relationship between grammars and metamodels (as discussed in “Toward an Engineering Discipline for Grammarware”, Klint, particularly section 2).

2.5 Benefits of and Current Challenges for MDE

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.

[Álvarez *et al.* 2001] José Álvarez, Andy Evans, and Paul Sammut. MML and the metamodel architecture. In *Proc. Workshop on Transformation in UML*, 2001.

[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 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.

[Brown *et al.* 1998] William J. Brown, Raphael C. Malveau, Hays W. McCormick III, and Thomas J. Mowbray. *Anti Patterns*. Wiley, 1998.

[Cervelle *et al.* 2006] Julien Cervelle, Rémi Forax, and Gilles Roussel. Tatoo: an innovative parser generator. In *Principles and Practice of Programming in Java*, pages 13–20. ACM, 2006.

[Ceteva 2008] Ceteva. XMF – the extensible programming language [online]. [Accessed 30 June 2008] Available at: `http://www.ceteva.com/xmf.html`, 2008.

[Chen & Chou 1999] J.Y.J. Chen and S.C. Chou. Consistency management in a process environment. *Systems and Software*, 47(2-3):105–110, 1999.

[Cicchetti *et al.* 2008] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Automating co-evolution in model-driven engineering. In *Proc. EDOC*, pages 222–231. IEEE Computer Society, 2008.

[Clark *et al.* 2008] Tony Clark, Paul Sammut, and James Willians. Superlanguages: Developing languages and applications with XMF [online]. [Accessed 30 June 2008] Available at: `http://www.ceteva.com/docs/Superlanguages.pdf`, 2008.

[Cleland-Huang *et al.* 2003] Jane Cleland-Huang, Carl K. Chang, and Mark Christensen. Event-based traceability for managing evolutionary change. *IEEE Transactions on Software Engineering*, 29(9):796–810, 2003.

[Costa & Silva 2007] M. Costa and A.R. da Silva. RT-MDD framework – a practical approach. In *Proc. ECMDA-FA Workshop on Traceability*, pages 17–26, 2007.

[Czarnecki & Helsen 2006] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646, 2006.

[Deursen *et al.* 2000] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[Deursen *et al.* 2007] Arie van Deursen, Eelco Visser, and Jos Warmer. Model-driven software evolution: A research agenda. In *Proc. Workshop on Model-Driven Software Evolution*, pages 41–49, 2007.

[Dig & Johnson 2006a] Danny Dig and Ralph Johnson. Automated upgrading of component-based applications. In *OOPSLA Companion*, pages 675–676, 2006.

[Dig & Johnson 2006b] Danny Dig and Ralph Johnson. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance and Evolution*, 18(2):83–107, 2006.

[Dig *et al.* 2006] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automated detection of refactorings in evolving components. In *Proc. European Conference on Object-Oriented Programming*, volume 4067 of *LNCS*, pages 404–428. Springer, 2006.

[Dig *et al.* 2007] Danny Dig, Kashif Manzoor, Ralph Johnson, and Tien N. Nguyen. Refactoring-aware configuration management for object-oriented programs. In *Proc. International Conference on Software Engineering*, pages 427–436. IEEE Computer Society, 2007.

[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.

[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.

[Frankel 2002] David Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing.* Wiley, 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.

[Graham 1993] Paul Graham. *On Lisp: Advanced Techniques for Common Lisp.* Prentice-Hall, 1993.

[Greenfield *et al.* 2004] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.* Wiley, 2004.

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

[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.* 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.* 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.

[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.

[JetBrains 2008] JetBrains. MPS – Meta Programming System [online]. [Accessed 30 June 2008] Available at: `http://www.jetbrains.com/mps/index.html`, 2008.

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

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

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

[Kelly & Tolvanen 2008] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modelling*. Wiley, 2008.

[Kerievsky 2004] Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 2004.

[Kleppe *et al.* 2003] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[Klint *et al.* 2003] P. Klint, R. Lämmel, and C. Verhoef. Towards an engineering discipline for grammarware. *ACM Transactions on Software Engineering Methodology*, 14:331–380, 2003.

[Kolovos *et al.* 2006a] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. 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.* 2006b] 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.

[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.

[Mens & Tourwé 2004] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[Mens *et al.* 2007] Tom Mens, Gabriele Taentzer, and Dirk Müller. Challenges in model refactoring. In *Proc. Workshop on Object-Oriented Reengineering*, 2007.

[Merriam-Webster 2010] Merriam-Webster. Definition of Nuclear Family. `http://www.merriam-webster.com/dictionary/nuclear%20family`, 2010.

[Moad 1990] J Moad. Maintaining the competitive edge. *Datamation*, 36(4):61–66, 1990.

[Moha *et al.* 2009] Naouel Moha, Vincent Mahé, Olivier Barais, and Jean-Marc Jézéquel. Generic model refactorings. In *Proc. MoDELS*, volume 5795 of *LNCS*, pages 628–643. Springer, 2009.

[Muller & Hassenforder 2005] Pierre-Alain Muller and Michel Hassenforder. HUTN as a Bridge between ModelWare and GrammarWare. In *Proc. Workshop in Software Modelling Engineering*, 2005.

[Nentwich *et al.* 2003] C. Nentwich, W. Emmerich, A. Finkelstein, and E. Ellmer. Flexible consistency checking. *ACM Transactions on Software Engineering and Methodology*, 12(1):28–63, 2003.

[Nguyen *et al.* 2005] Tien Nhut Nguyen, Cheng Thao, and Ethan V. Munson. On product versioning for hypertexts. In *Proc. International Workshop on Software Configuration Management (SCM)*, pages 113–132. ACM, 2005.

[Oldevik *et al.* 2005] Jon Oldevik, Tor Neple, Roy Grønmo, Jan Øyvind Aagedal, and Arne-Jørgen Berre. Toward standardised model to text transformations. In *Proc. ECMDA-FA*, volume 3748 of *LNCS*, pages 239–253. Springer, 2005.

[Olsen & Oldevik 2007] Gøran K. Olsen and Jon Oldevik. Scenarios of traceability in model to text transformations. In *Proc. ECMDA-FA*, volume 4530 of *Lecture Notes in Computer Science*, pages 144–156. Springer, 2007.

[OMG 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. 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.

[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.

[Stahl *et al.* 2006] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, 2006.

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

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

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

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

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

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

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

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

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

[Ward 1994] Martin P. Ward. Language-oriented programming. *Software — Concepts and Tools*, 15(4):147–161, 1994.

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

[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.