Evolution in Model-Driven Engineering

Louis M. Rose

August 18, 2010

Contents

Chapter 2

Background

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

2.1 MDE Terminology and Principles

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

2.1.1 Models

Models are fundamental to MDE. [Kurtev 2004] identifies many definitions of the term model, such as: "any subject using a system A that is neither directly nor indirectly interacting with a system B to obtain information about the system B, is using A as a model for B." [Apostel 1960], "a model is a representation of a concept. The representation is purposeful and used to abstract from reality the irrelevant details." [Starfield *et al.* 1990], and "a model is a simplification of a system written in a well-defined language." [Bézivin & Gerbé 2001].

While there are many definitions of the term model, a common notion is that a model is a representation of the real-world [Kurtev 2004, pg12]. The part of the real-world represented by a model is termed the *domain*, the *object system* or, simply the *system*. A further commonality is noted by [Kolovos *et al.* 2006c]: a model may have either a textual or graphical representation.

[Ackoff 1962] defines *analogous* models as those which share some characteristics and can be used in place of their object system. An aeroplane toy that can fly is an analogous model of an aeroplane. In computer science, models can be used to construct a computer system. A model of an object system, say the lending service of a library, might be used to decide the way in which data is stored on disk, or the way in which a program is to be structured.

[Jackson 1995] proposes that the models constructed in computer science are analogous to two systems: the object system (e.g. the library lending service in the real-world) and the computer system (e.g. the combination of software and hardware used to implement a library lending service). A model can be used to think about both the real system and the computer system. Figure 2.1 illustrates this notion further. According to [Jackson 1995], a model is both the description of the domain (object system) and the machine (computer system). Computer scientists switch between *designations* when using a model to think about the object system or to think about the software system.

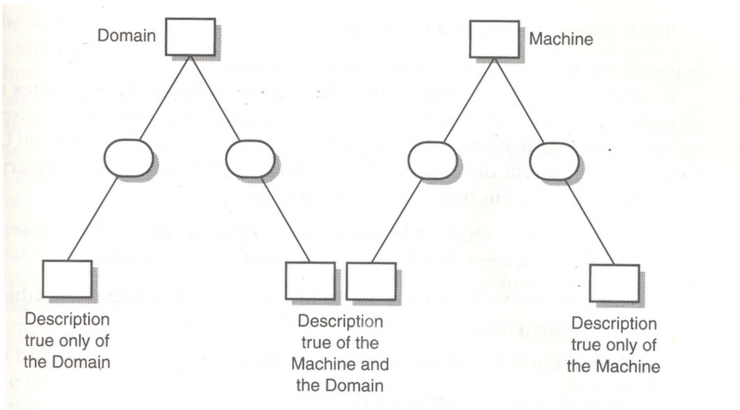

Figure 2.1: Jackson's definition of a model, taken from [Jackson 1995, pg.125].

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

2.1.2 Modelling languages

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

Conformance is a relationship between a metamodel and a model. A model *conforms to* a metamodel when the metamodel specifies every concept used in the model definition, and the model uses the metamodel concepts according to the rules specified by the metamodel. Conformance can be described by a set of constraints between models and metamodels [Paige *et al.* 2007]. When all constraints are satisfied, a model conforms to a metamodel. For example, a conformance constraint might state that every object in the model has a corresponding non-abstract class in the metamodel.

Metamodels facilitate model interchange and, therefore, interoperability between modelling tools. For this reason, Evans recommends that software engineers "use a well-documented shared language that can express the necessary domain information as a common medium of communication." [Evans 2004, pg377]. To support this recommendation, Evans discusses the way in which chemists have collaborated to define a standardised language for describing chemical structures, Chemical Markup Language (CML)[1]. The standardisation of CML has facilitated interoperability between tools for specification, analysis and simulation.

A metamodel typically comprises three categories of constraint:

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

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

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

2.1.3 MOF: A metamodelling language

Software engineers using MDE can use existing and define new metamodels. To facilitate interoperability between MDE tools, the OMG has standardised a lan-

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

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

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

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

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

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

2.1.4 Model Management

In MDE, models are *managed* to produce software. [Melnik 2004] first described *model management* as a collection of operators for manipulating models. [Kolovos 2009] explores a means for increasing the interoperability of model management operations. This thesis uses the term *model management* to refer to development activities that manipulate models for the purpose of producing software. Model management activities typical in MDE, such as model transformation and validation, are discussed in this section. Section 2.2 discusses MDE

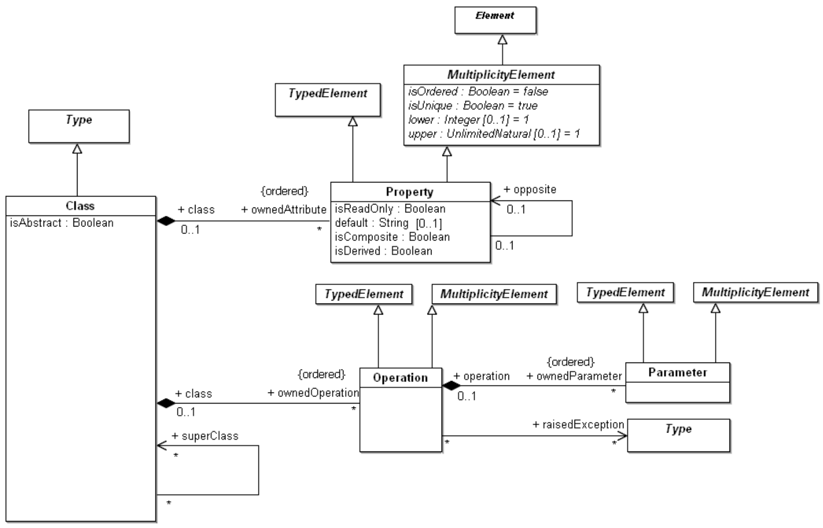

Figure 2.2: A fragment of the UML metamodel defined in MOF, from [OMG 2007a].

guidelines and methods, and describes the way in which model management activities are used together to produce software in MDE.

Model Transformation

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

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

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

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

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

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

Model-to-Model (M2M) Transformation M2M transformation is used to derive models from others. By automating the derivation of models from others, M2M transformation has the potential to reduce the cost of engineering large and complex systems that can be represented as a set of interdependent models [Sendall & Kozaczynski 2003].

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

Many M2M transformation languages have been proposed, such as the Atlas Transformation Language (ATL) [Jouault & Kurtev 2005], the Epsilon Transformation Language (ETL) [Kolovos *et al.* 2008a] and VIATRA [Varró & Balogh 2007]. The OMG [OMG 2008c] provide a standardised M2M transformation language, Queries/Views/Transformations (QVT) [OMG 2005]. M2M transformation languages can be categorised according to their *style*, which is either declarative, imperative or hybrid.

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

Because declarative M2M transformation languages cannot be used to solve some categories of transformation problem [Patrascoiu & Rodgers 2004] and imperative M2M transformation languages are argued to be difficult to write and maintain [Kolovos 2009, pg.45], [Kolovos *et al.* 2008a] notes that the current consensus is that hybrid languages, such as ATL are more suitable for specifying model transformation than pure imperative or declarative languages.

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

```
rule Machine2Package
```

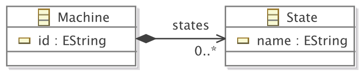

Figure 2.3: Exemplar State Machine metamodel.

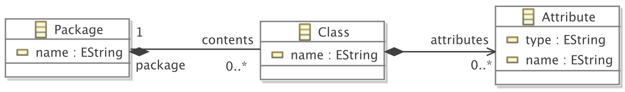

Figure 2.4: Exemplar Object-Oriented metamodel.

```
2    transform m : StateMachine!Machine
3    to      p : ObjectOriented!Package {
4
5    p.name   := 'uk.ac.york.cs.' + m.id;
6    p.contents := m.states.equivalent();
7  }
8
9  rule State2Class
10   transform s : StateMachine!State
11   to      c : ObjectOriented!Class
12
13   guard: not s.isFinal {
14
15   c.name := s.name + 'State';
16 }
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Xtext[5] and EMFtext [Heidenreich *et al.* 2009] 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.

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

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

and five `States` with the `names`, `stationary`, `up`, `down`, `stopping`, and `emergency`.

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

Listing 2.3: Exemplar T2M transformation in EMFtext

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

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

Model Validation

Model validation provides a mechanism for managing the integrity of the software developed using MDE. A model that omits information is said to be *incomplete*, while related models that suggest differences in the underlying phenomena are said to be *contradicting* [Kolovos 2009]. Incompleteness and contradiction are two examples of *inconsistency*. In MDE, inconsistency is detrimental, because, when artefacts are automatically derived from each other, the inconsistency of one artefact might be propagated to others. Model validation is used to detect, report and reconcile inconsistency throughout a MDE process.

[Kolovos 2009] observes that inconsistency detection is inherently pattern-based and, hence, higher-order languages are more suitable for model validation than so-called "third-generation" programming languages (such as Java). The Object Constraint Language (OCL) [OMG 2006] is an OMG standard that can be used to specify consistency constraints on UML and MOF models. OCL cannot be used to specify inter-model constraints, unlike the xlinkit toolkit [Nentwich *et al.* 2003] and the Epsilon Validation Language (EVL) [Kolovos *et al.* 2008b].

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

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

Listing 2.4: Exemplar model validation in the Epsilon Validation Language

Further model management activities

In addition to model transformation and validation, further examples of model management activities include model comparison (e.g. [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. [Kolovos *et al.* 2006a]), in which two or more models are combined to produce a unified model.

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

2.1.5 Summary

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

2.2 MDE Guidelines and Methods

For performing MDE, new engineering practices and processes have been proposed. Proponents of MDE have produced guidance and methods for MDE. This section discusses the guidance for MDE set out in the Model-Driven Architecture [OMG 2008b] and the methods for MDE described in [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 guidelines for MDE. For instance, MDA prescribes the use of a Platform Independent Model (PIM) and one or more Platform Specific Models (PSMs).

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

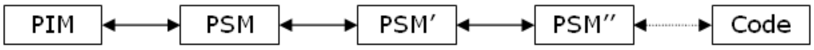

Figure 2.5: Interactions between a PIM and several PSMs.

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

Standards for MDA

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

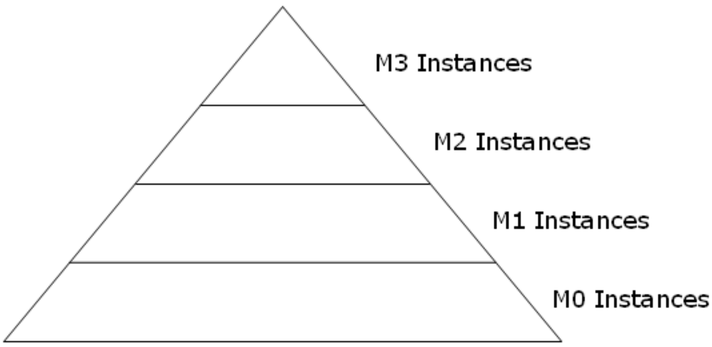

Figure 2.6: The tiers of standards used as part of MDA.

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

Interpretations of MDA

[McNeile 2003] identifies two ways in which engineers have interpreted MDA. Both interpretations begin with a PIM, but the way in which executable code is produced varies:

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

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

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

The difference between translationist and elaborationist approaches to MDE is related to a difference in the way in which models are viewed in traditional approaches to software engineering. For example, [Evans 2004] proposes the use of models throughout the development process, and the way in which code is structured is driven by the model. By contrast, [Martin & Martin 2006, ch14] prescribes modelling only for communicating and reasoning about a design, and not "as a long-term replacement for real, working software". Rather [Martin & Martin 2006] advocates using models to quickly compare different ways in which a system might be structured and then to disregard those models in favour of working code.

2.2.2 Methods for MDE

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

Architecture-Centric Model-Driven Software Development

Model-Driven Software Development is the term given to MDE by in [Stahl *et al.* 2006]. The style of MDE that [Stahl *et al.* 2006] describes, *architecture-centric model-driven software development* (AC-MDSD), focuses on generating the infrastructure of large-scale applications. For example, a typical J2EE application contains concepts (such as EJBs, descriptors, home and remote interfaces) that "admittedly contain domain-related information such as method signatures, but which also exhibit a high degree of redundancy" [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 Domain-Specific Modelling (DSM), a collection of principles, practices and advice for constructing systems using MDE. DSM is based on the translationist interpretation of MDA: domain models are transformed directly to code. In motivating the need for DSM, Kelly and Tolvanen state that large productivity gains were made when third-generation programming languages were used in place of assembler, and that no paradigm shift has since been able to replicate this degree of improvement. Tolvanen[6] notes that DSM focuses on increasing the productivity of software engineering by allowing developers to specify solutions by using models that describe the application domain.

To perform DSM, expert developers define:

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

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

Tools for defining domain-specific modelling languages, editors and code generators enable DSM [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 termed *language-oriented programming* (LOP), which also requires tools to simplify the specification of new languages. LOP is discussed further in Section 2.4.

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

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

Microsoft Software Factories

[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, seeks to address problems with economies of scope in software engineering by borrowing concepts from product-line engineering. [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 choose values for the variation points in the product. To simplify the creation of these models, Software Factories propose model creation wizards. Greenfield et al. state that "moving from totally-open ended hand-coding to more constrained forms of specification [such as wizard-based feature selection] are the key to accelerating software development" [Greenfield *et al.* 2004, pg179]. By providing explanations that assist in making decisions, the wizards used in Software Factories guide users towards best practices for customising a product.

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

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

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

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

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

2.3 MDE Tools

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

Section 2.3.1 provides an overview of the Eclipse Modelling Framework (EMF) [Eclipse 2008a], which implements MOF and underpins many contemporary MDE tools and languages, facilitating their interoperability. Section 2.3.2 discusses Epsilon [Eclipse 2008c], an extensible platform for the specification of model management languages. The highly extensible nature of Epsilon (which is described below) makes it an ideal host for the rapid prototyping of languages and exploring research hypotheses.

The purpose of this section is to review EMF and Epsilon, which are used throughout the remainder of the thesis, and not to provide a thorough review of all MDE tools. There are many other MDE tools and environments that this section does not discuss, such as ATL [ATLAS 2007] and VIATRA [Varró & Balogh 2007] for M2M transformation, oAW [openArchitectureWare 2007] for model transformation and validation, MOFScript [Oldevik *et al.* 2005] and XPand [openArchitectureWare 2008] for M2T transformation, and the AMMA [INRIA 2007] platform for large-scale modelling, model weaving and software modernisation.

2.3.1 Eclipse Modelling Framework (EMF)

[Eclipse 2008b] is an open-source community seeking 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 is used to generate metamodel-specific editors for loading, storing and constructing models. EMF model editors comprise a navigation view for specifying the elements of the model, and a properties view for specifying the features of model elements. Figure 2.7 shows an EMF model editor for a simplistic state machine language. The navigation (or tree) view is shown in the top pane, while the properties view is shown in the bottom pane.

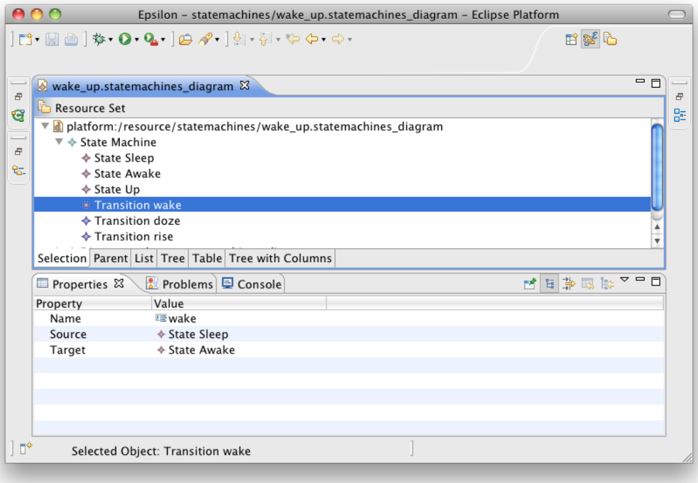

Figure 2.7: EMF state machine model editor.

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

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

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

Many MDE tools are interoperable with EMF, enriching its functionality. The remainder of this section discusses one tool that is interoperable with EMF,

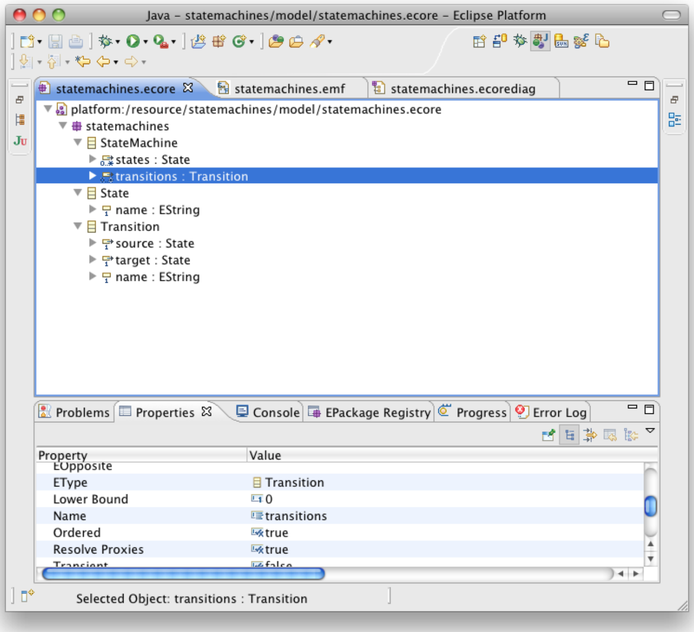

Figure 2.8: EMF's tree-based metamodel editor.

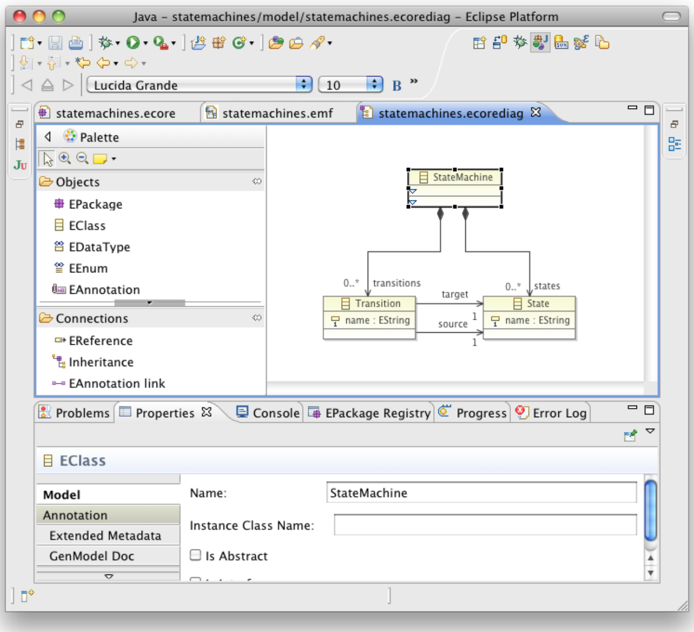

Figure 2.9: EMF's graphical metamodel editor.

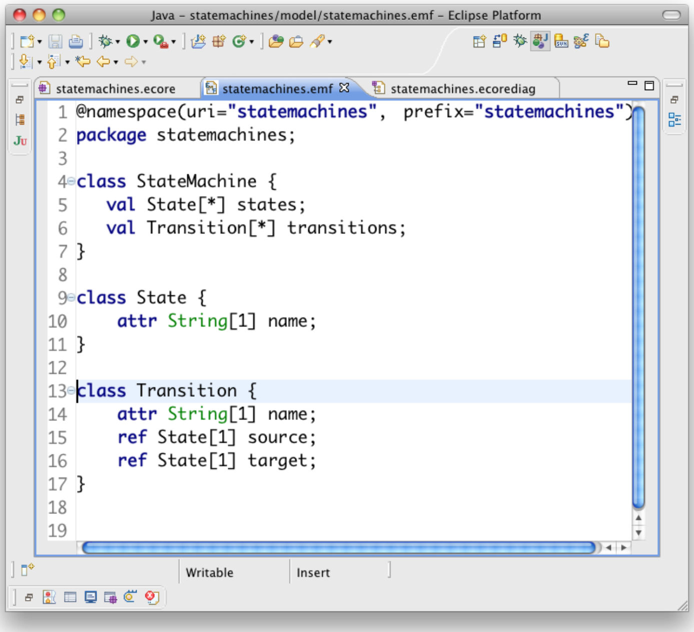

Figure 2.10: The Emfatic textual metamodel editor for EMF.

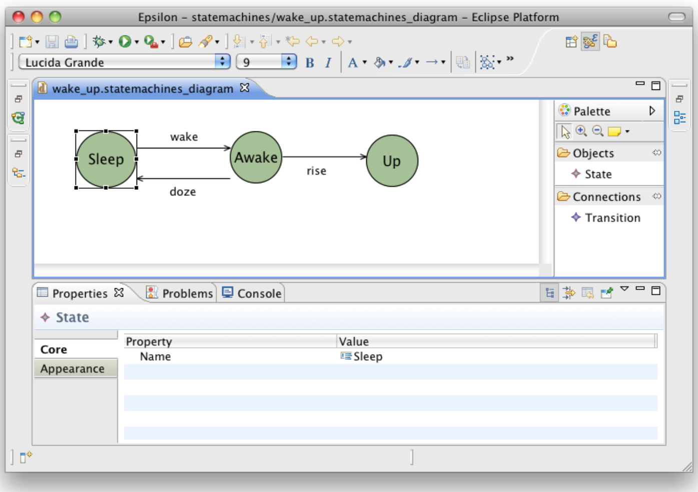

Figure 2.11: GMF state machine model editor.

Epsilon, which is a suitable platform for rapid prototyping of model management languages and, hence, is useful for performing MDE research.

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 2.12 illustrates the various components of Epsilon.

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

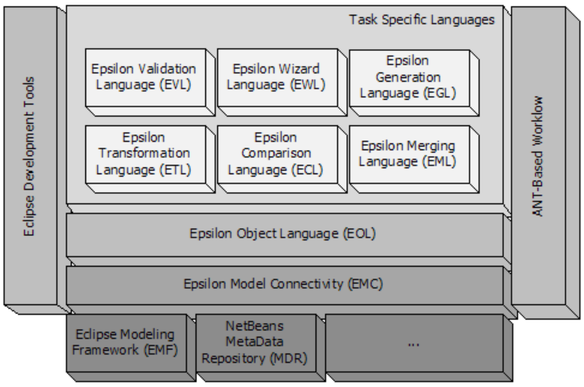

Figure 2.12: The architecture of Epsilon, taken from [Rose *et al.* 2008].

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 M2T transformation. Epsilon has been used extensively for the work described in Chapter 5.

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

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

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

Epsilon is a member of the Eclipse GMT [Eclipse 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.

2.3.3 Summary

This section has introduced the MDE tools used throughout the remainder of the thesis. The Eclipse Modeling Framework (EMF) provides an implementation of MOF, Ecore, for defining metamodels. From metamodels defined in Ecore, EMF can generate code for metamodel-specific editors and for persisting models to disk. EMF is arguably the most widely used contemporary MDE modelling framework and its functionality is enhanced by numerous tools, such as the Graphical Modeling Framework (GMF) and Epsilon. GMF allows metamodel developers to specify a graphical concrete syntax for metamodels, and can be used to generate graphical model editors. Epsilon is an extensible platform for defining and executing model management languages, provides a high degree of re-use for defining new model management languages and can be used with a range of modelling frameworks, including EMF.

2.4 Research Relating to MDE

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

2.4.1 Domain-Specific Languages

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

Like modelling languages, DSLs describe abstract syntax. Furthermore, a common language can be used to define DSLs (e.g. EBNF [ISO/IEC 1996]), like the use of MOF for defining modelling languages. In addition to abstract syntax, DSLs typically define a textual concrete syntax but, like modelling languages, can utilise a graphical concrete syntax.

Cobol, Fortran and Lisp first existed as DSLs for solving problems in the domains of business processing, numeric computation and symbolic processing respectively, and evolved to become general-purpose programming languages [Deursen *et al.* 2000]. SQL, on the other hand, is an example of a DSL that, despite undergoing much change, has not grown into a general-purpose language. Unlike a general-purpose language, a single DSL cannot be used to program an entire application. DSLs are often small languages at inception, but can grow to become complicated (such as SQL). Within their domain, DSLs should be easy to read, understand and edit [Fowler 2005].

There are two ways in which DSLs are typically implemented. An *internal* DSL is implemented by describing the domain using constructs from a general-purpose language (the *host*) [Dmitriev 2004, Fowler 2010]. Examples of internal DSLs include the frameworks for working with collections that are included in some programming languages (e.g. STL for C++, the Collections API for Java). Some languages are better than others for hosting internal DSLs. For example, [Fowler 2005] proposes Ruby as a suitable host for DSLs due to its "unintrusive syntax and flexible runtime evaluation." [Graham 1993] describes a technique for implementing internal DSLs in Lisp, in which macros are used to translate domain-specific concepts to Lisp abstractions.

[Dmitriev 2004] reports that internal DSLs can exhibit some unsatisfactory characteristics because there is often a mismatch between domain and programming abstractions. For this reason, [Dmitriev 2004] prefers to implement DSLs by *translating* DSL programs into code written in a general-purpose language. [Fowler 2010] uses the term *external* for this style of DSL implementation. Programs written in simple DSLs are often easy to translate to programs in an existing general-purpose language [Parr 2007]. Approaches to translation include preprocessing; building or generating an interpreter or compiler; or extending an existing compiler or interpreter [Dmitriev 2004].

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

Internal and external DSLs have been successfully used as part of application development in many domains, as described in [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 (such as automation of system deployment and configuration). More recently, some developers are building complete applications by combining DSLs, in a style of development called Language-Oriented Programming.

2.4.2 Language-Oriented Programming

[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 combine programs written in small (domain-specific) languages (such as awk, make, sed, lex and yac) to solve problems. Lisp, Smalltalk and Ruby programmers often construct domain-specific languages when developing programs [Graham 1993, Fowler 2005].

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

A *language workbench* [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]).

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

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

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

2.4.3 Summary

This section has described two areas of research related to MDE, domain-specific languages (DSLs) and language-oriented programming (LOP). DSLs facilitate the encoding of solutions for a particular problem domain. For solving problems in their domain, DSLs can be easier to read, use and edit than general-purpose programming languages [Deursen *et al.* 2000, Fowler 2010]. During MDE, one or more DSLs may be used to model the domain, and the tools and techniques for implementing DSLs can be used for MDE.

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

2.5 Benefits of and Current Challenges for MDE

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

2.5.1 Benefits

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

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

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

System evolution The guidelines set out for MDE in MDA [OMG 2008b] highlight principles and patterns for modelling to increase the adaptability of software systems by, for example, separating platform-specific and platform-independent detail. When the target platform changes (for example a new technological architecture is required), only part of the system needs to be changed. The platform-independent detail can be re-used wholesale.

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

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

2.5.2 Challenges

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

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

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

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

Development artefact evolution Notwithstanding the benefits of MDE for managing the evolution of systems, the introduction of additional development artefacts (such as models and metamodels) and activities (such as model management) presents additional challenges for the way in which developers manage software evolution [Mens & Demeyer 2007]. For example, in traditional approaches to software engineering, maintainability is primarily achieved by restructuring code, updating documentation and regression testing [Feathers 2004].

It is not yet clear the extent to which existing maintenance activities can be applied in MDE. (For example, should models be tested and, if so, how?)

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

2.5.3 Summary

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

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

2.6 Chapter Summary

To be completed.

Bibliography

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

[Ackoff 1962] Russell L. Ackoff. *Scientific Method: Optimizing Applied Research Decisions.* John Wiley and Sons, 1962.

[Aizenbud-Reshef *et al.* 2005] N. Aizenbud-Reshef, R.F. Paige, J. Rubin, Y. Shaham-Gafni, and D.S. Kolovos. Operational semantics for traceability. In *Proc. ECMDA-FA Workshop on Traceability*, pages 8–14, 2005.

[Alexander *et al.* 1977] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series).* Oxford University Press, 1977.

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

[Apostel 1960] Leo Apostel. Towards the formal study of models in the non-formal sciences. *Synthese*, 12:125–161, 1960.

[Arendt *et al.* 2009] Thorsten Arendt, Florian Mantz, Lars Schneider, and Gabriele Taentzer. Model refactoring in eclipse by LTK, EWL, and EMF Refactor: A case study. In *Proc. Joint MoDSE-MCCM Workshop*, 2009.

[ATLAS 2007] ATLAS. Atlas Transformation Language Project Website [online]. [Accessed 30 June 2008] Available at: `http://www.eclipse.org/m2m/atl/`, 2007.

[Backus 1978] John Backus. The history of FORTRAN I, II and III. *History of Programming Languages*, 1:165–180, 1978.

[Balazinska *et al.* 2000] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lagüe, and Kostas Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proc. Working Conference on Reverse Engineering*, pages 98–107. IEEE Computer Society, 2000.

[Banerjee *et al.* 1987] Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *Proc. Special Interest Group on Management of Data*, volume 16, pages 311–322. ACM, 1987.

[Beck & Cunningham 1989] Kent Beck and Ward Cunningham. Constructing abstractions for object-oriented applications. *Journal of Object Oriented Programming*, 2, 1989.

[Bézivin & Gerbé 2001] Jean Bézivin and Olivier Gerbé. Towards a precise definition of the OMG/MDA framework. In *Proc. ASE*, pages 273–280. IEEE Computer Society, 2001.

[Bézivin 2005] Jean Bézivin. On the unification power of models. *Software and System Modeling*, 4(2):171–188, 2005.

[Biermann *et al.* 2006] Enrico Biermann, Karsten Ehrig, Christian Köhler, Günter Kuhns, Gabriele Taentzer, and Eduard Weiss. Emf model refactoring based on graph transformation concepts. *ECEASST*, 3, 2006.

[Bloch 2005] Joshua Bloch. How to design a good API and why it matters [online]. Keynote address to the LCSD Workshop at OOPSLA, October 2005, San Diego, United States of America. [Accessed 23 July 2009] Available at: `http://lcsd05.cs.tamu.edu/slides/keynote.pdf`, 2005.

[Bohner 2002] Shawn A. Bohner. Software change impacts - an evolving perspective. In *Proc. International Conference on Software Maintenance (ICSM)*, pages 263–272. IEEE Computer Society, 2002.

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

[Briand *et al.* 2003] Lionel C. Briand, Yvan Labiche, and L. O'Sullivan. Impact analysis and change management of uml models. In *Proc. International Conference on Software Maintenance (ICSM)*, pages 256–265. IEEE Computer Society, 2003.

[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.

[Feathers 2004] Michael C. Feathers. *Working Effectively with Legacy Code*. Prentice Hall, 2004.

[Ferrandina *et al.* 1995] Fabrizio Ferrandina, Thorsten Meyer, Roberto Zicari, Guy Ferran, and Joëlle Madec. Schema and database evolution in the O2 object database system. In *Very Large Data Bases*, pages 170–181. Morgan Kaufmann, 1995.

[Fowler 1999] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[Fowler 2002] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[Fowler 2005] Martin Fowler. Language workbenches: The killer-app for domain specific languages? [online]. [Accessed 30 June 2008] Available at: `http://www.martinfowler.com/articles/languageWorkbench.html`, 2005.

[Fowler 2010] Martin Fowler. *Domain Specific Languages.* Addison-Wesley Professional, 2010.

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

[Frenzel 2006] Leif Frenzel. The language toolkit: An API for automated refactorings in eclipse-based IDEs [online]. [Accessed 02 August 2010] Available at: `http://www.eclipse.org/articles/Article-LTK/ltk.html`, 2006.

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

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

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

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

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

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

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

[Gronback 2009] R.C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit.* Addison-Wesley Professional, 2009.

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

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

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

[Heidenreich *et al.* 2009] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and refinement of textual syntax for models. In *Proc. ECMDA-FA*, volume 5562 of *LNCS*, pages 114–129. Springer, 2009.

[Herrmannsdoerfer *et al.* 2008] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Juergens. Automatability of coupled evolution of metamodels and models in practice. In *Proc. International Conference on Model Driven Engineering Languages and Systems*, volume 5301 of *LNCS*, pages 645–659. Springer, 2008.

[Herrmannsdoerfer *et al.* 2009a] M. Herrmannsdoerfer, D. Ratiu, and G. Wachsmuth. Language evolution in practice. In *Proc. SLE*, volume 5696 of *LNCS*, pages 3–22. Springer, 2009.

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

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

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

[INRIA 2007] INRIA. AMMA project page [online]. [Accessed 30 June 2008] Available at: `http://wiki.eclipse.org/AMMA`, 2007.

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

[ISO/IEC 1996] Information Technology ISO/IEC. Syntactic metalanguage – Extended BNF. ISO 14977:1996 International Standard, 1996.

[ISO/IEC 2002] Information Technology ISO/IEC. Z Formal Specification Notation – Syntax, Type System and Semantics. ISO 13568:2002 International Standard, 2002.

[Jackson 1995] M. Jackson. *Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices.* ACM Press, 1995.

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

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

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

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

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

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

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

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

[Kolovos *et al.* 2006a] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Merging models with the epsilon merging language (eml). In *Proc. MoDELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 215–229. Springer, 2006.

[Kolovos *et al.* 2006b] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Model comparison: a foundation for model composition and model transformation testing. In *Proc. Workshop on Global Integrated Model Management*, pages 13–20, 2006.

[Kolovos *et al.* 2006c] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. The Epsilon Object Language (EOL). In *Proc. ECMDA-FA*, volume 4066 of *LNCS*, pages 128–142. Springer, 2006.

[Kolovos *et al.* 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.

[Kurtev 2004] Ivan Kurtev. *Adaptability of Model Transformations.* PhD thesis, University of Twente, Netherlands, 2004.

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

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

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

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

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

[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.

[Melnik 2004] Sergey Melnik. *Generic Model Management: Concepts and Algorithms.* PhD thesis, University of Leipzig, Germany, 2004.

[Mens & Demeyer 2007] Tom Mens and Serge Demeyer. *Software Evolution.* Springer-Verlag, 2007.

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

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

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

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

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

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

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

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

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

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

[OMG 2001] OMG. Unified Modelling Language 1.4 Specification [online]. [Accessed 15 September 2008] Available at: `http://www.omg.org/spec/UML/1.4/`, 2001.

[OMG 2004] OMG. Human-Usable Textual Notation 1.0 Specification [online]. [Accessed 30 June 2008] Available at: `http://www.omg.org/technology/documents/formal/hutn.htm`, 2004.

[OMG 2005] OMG. MOF QVT Final Adopted Specication [online]. [Accessed 22 July 2009] Available at: `www.omg.org/docs/ptc/05-11-01.pdf`, 2005.

[OMG 2006] OMG. Object Constraint Language 2.0 Specification [online]. [Accessed 30 June 2008] Available at: `http://www.omg.org/technology/documents/formal/ocl.htm`, 2006.

[OMG 2007a] OMG. Unified Modelling Language 2.1.2 Specification [online]. [Accessed 30 June 2008] Available at: `http://www.omg.org/spec/UML/2.1.2/`, 2007.

[OMG 2007b] OMG. Unified Modelling Language 2.2 Specification [online]. [Accessed 5 March 2010] Available at: `http://www.omg.org/spec/UML/2.2/`, 2007.

[OMG 2007c] OMG. XML Metadata Interchange 2.1.1 Specification [online]. [Accessed 30 June 2008] Available at: `http://www.omg.org/technology/documents/formal/xmi.htm`, 2007.

[OMG 2008a] OMG. Meta-Object Facility [online]. [Accessed 30 June 2008] Available at: `http://www.omg.org/mof`, 2008.

[OMG 2008b] OMG. Model Driven Architecture [online]. [Accessed 30 June 2008] Available at: `http://www.omg.org/mda/`, 2008.

[OMG 2008c] OMG. Object Management Group home page [online]. [Accessed 30 June 2008] Available at: `http://www.omg.org`, 2008.

[Opdyke 1992] William F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, IL, USA, 1992.

[openArchitectureWare 2007] openArchitectureWare. openArchitectureWare Project Website [online]. [Accessed 30 June 2008] Available at: `http://www.eclipse.org/gmt/oaw/`, 2007.

[openArchitectureWare 2008] openArchitectureWare. XPand Language Reference [online]. [Accessed 18 August 2010] Available at: `http://wiki.eclipse.org/AMMA`, 2008.

[Paige *et al.* 2007] Richard F. Paige, Phillip J. Brooke, and Jonathan S. Ostroff. Metamodel-based model conformance and multiview consistency checking. *ACM Transactions on Software Engineering and Methodology*, 16(3), 2007.

[Paige *et al.* 2009] Richard F. Paige, Louis M. Rose, Xiaocheng Ge, Dimitrios S. Kolovos, and Phillip J. Brooke. FPTC: Automated safety analysis for domain-specific languages. In *MoDELS Workshops and Symposia*, volume 5421 of *Lecture Notes in Computer Science*, pages 229–242. Springer, 2009.

[Parr 2007] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages.* Pragmatic Programmers, 2007.

[Patrascoiu & Rodgers 2004] Octavian Patrascoiu and Peter Rodgers. Embedding OCL expressions in YATL. In *Proc. OCL and Model-Driven Engineering Workshop*, 2004.

[Pilgrim *et al.* 2008] Jens von Pilgrim, Bert Vanhooff, Immo Schulz-Gerlach, and Yolande Berbers. Constructing and visualizing transformation chains. In *Proc. European Conference on the Model Driven Architecture – Foundations and Applications*, volume 5095 of *LNCS*, pages 17–32. Springer, 2008.

[Pizka & Jürgens 2007] M. Pizka and E. Jürgens. Automating language evolution. In *Proc. Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 305–315. IEEE Computer Society, 2007.

[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, Markus Herrmannsdoerfer, James R. Williams, Dimitrios S. Kolovos, Kelly Garcés, Richard F. Paige, and Fiona A.C. Polack. A comparison of model migration tools. In *Proc. MoDELS*, volume TBC of *Lecture Notes in Computer Science*, page TBC. Springer, 2010.

[Rose *et al.* 2010b] Louis M. Rose, Dimitrios S. Kolovos, Nicholas Drivalos, James. R. Williams, Richard F. Paige, Fiona A.C. Polack, and Kiran J. Fernandes. Concordance: An efficient framework for managing model integrity [submitted to]. In *Proc. European Conference on Modelling Foundations and Applications*, 2010.

[Rose *et al.* 2010c] Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Migrating activity diagrams with epsilon flock. In *Proc. TTC*, 2010.

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

[Rose *et al.* 2010e] Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Model migration with epsilon flock. In *Proc. ICMT*, volume 6142 of *Lecture Notes in Computer Science*, pages 184–198. Springer, 2010.

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

[Selic 2005] Bran Selic. Whats new in UML 2.0? *IBM Rational software*, 2005.

[Sendall & Kozaczynski 2003] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20:42–45, 2003.

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

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

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

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

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

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

[Starfield *et al.* 1990] M. Starfield, K.A. Smith, and A.L. Bleloch. *How to model it: Problem Solving for the Computer Age.* McGraw-Hill, 1990.

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

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

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

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

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

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

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

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

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

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

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

[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.