# Chapter 4

# Analysis

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

Figure 4.1 summarises the objectives of this chapter. Examples of evolution in MDE projects were located (Section 4.1) and used to analyse existing co-evolution techniques. Analysis led to a categorisation and comparison of existing co-evolution approaches (Section 4.2) and to the identification of modelling framework characteristics that restrict the way in which co-evolution can be managed (Section 4.2.1). Research requirements for this thesis were identified from the analysis presented in this chapter (Section 4.3).

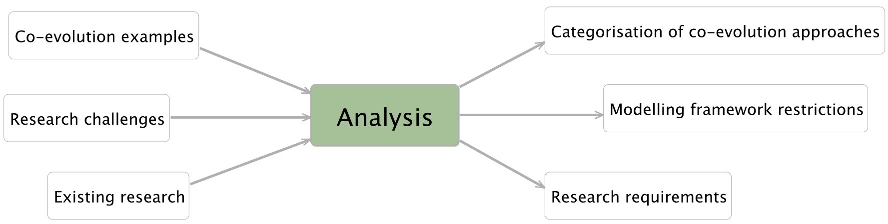



Figure 4.1: Analysis chapter overview.

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

## 4.1 Locating Data

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

### 4.1.1 Requirements

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

#### Common requirements

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

#### Co-evolution requirements

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

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

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

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

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

#### Synchronisation requirements

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

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

### 4.1.2 Project Selection

Eight candidate projects were considered for the study. Table 4.1.2 shows which of the requirements are fulfilled by each of the candidates. Each candidate is now discussed in turn.

#### GSN

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

### OMG

The Object Management Group (OMG) [OMG 2008c] oversees the development of model-driven technologies. The Vice President and Technical Director of OMG, Andrew Watson, references the development of two MDE projects in [Watson 2008]. Personal correspondence with Watson ascertained that source code is available for one of the projects, but there is no version history. **Suitability for study:** Unsuitable.

### Zoos

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

### MDT

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

### MODELPLEX

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

### FPTC

Failure Propagation and Transformation Calculus (FPTC), developed by Malcolm Wallace in this department, provides a means for reasoning about the failure behaviour of complex systems. In an earlier project, Richard Paige and the author developed an implementation of FPTC in Eclipse. The implementation includes an FPTC metamodel. More recent work with Philippa

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

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

Conmy, a Research Associate in this department, has identified a significant flaw in the implementation, leading to changes to the metamodel. The metamodel changes affected the conformance of existing FPTC models. Conmy has made available copies of FPTC models from before and after the changes. **Suitability for study:** Suitable for studying co-evolution. Unsuitable for studying synchronisation, because, although the tool includes a transformation, the target models are produced as output from a simulation, never stored and hence do not become inconsistent with their source model.

### xText

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

### GMF

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

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

GMF fulfils almost all of the requirements for the study. Co-evolution data is available, including migration strategies. The GMF source code repository does not contain examples of the kinds of change that cause inconsistency between the models (R8). **Suitability for study:** Suitable for studying co-evolution. Unsuitable for studying synchronisation.

### Summary of selection

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

[4] Private communication, 2008.

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

### 4.1.3 Other examples

Few MDE projects fulfilled all of the requirements for studying evolution, so additional data was sought from alternative sources. Examples were located from object-oriented systems – which have some similarities to systems developed using MDE – and via collaboration with colleagues on two projects, both of which involved developing a system using MDE.

#### Examples of evolution from object-oriented systems

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

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

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

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

2. **Navigational refactorings** convert, for example, between bi-directional and uni-directional associations. These changes are often non-breaking

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

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

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

Figure 4.2 illustrates a refactoring that changes a reference object to a value object [Fowler 1999][pg183]. Value objects are immutable, and cannot be shared (i.e. any two objects cannot refer to the same value object). By contrast, reference objects are mutable, and can be shared. Figure 4.2 indicates that applying the refactoring restricts the multiplicity of the association (on the Order end) to 1 (implied by the composition); prior to the refactoring the multiplicity is many-valued.

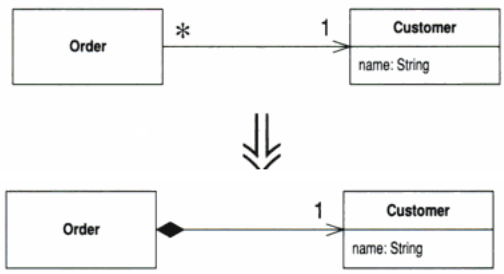



Figure 4.2: Refactoring a reference to a value. Taken from [Fowler 1999][pg183].

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

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

```

  delete c
next c
```

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

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

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

**Research collaborations**

As well as the example data located from object-oriented system, collaboration on projects using MDE with two colleagues provided several examples of evolution. A graphical editor for process-oriented programs was developed with Adam Sampson, then a Research Associate at the University of Kent, and is described in Appendix B. Additionally, the feasibility of a tool for generating story-worlds for interactive narratives was investigated with Heather Barber, then a postdoctoral researcher in this department.

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

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

### 4.1.4 Summary

This section has described the identification of example data for analysing the way in which evolution is identified and managed in the context of MDE. Example data was sought from existing MDE projects, a related domain (refactoring of object-oriented systems) and collaborative work on MDE projects (with Sampson and Barber). Eight MDE projects were located, three of which satisfied the requirements for a study of co-evolutionary changes in the context of model-driven engineering. One of the three projects, GMF, was reserved for the evaluation presented in Chapter 6. Refactorings of object-oriented programming supplemented the data available from the existing MDE projects. Collaboration with Sampson and Barber yielded further examples of co-evolution.

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

## 4.2 Analysing Existing Techniques

The examples of co-evolution identified in the previous section were analysed to identify and compare existing techniques for managing co-evolution. This section discusses the results of analysing the examples; namely a deeper understanding of modelling framework characteristics that affect the management of co-evolution (Section 4.2.1), and a categorisation of existing techniques for managing co-evolution (Sections 4.2.2 and 4.2.3) . The work presented here was published in [Rose *et al.* 2009b, Rose *et al.* 2009a].

### 4.2.1 Modelling Framework Characteristics

Analysis of the co-evolution examples identified above highlighted characteristics of modern MDE modelling development environments that affect the way in which co-evolution can be managed.

#### Model-Metamodel Separation

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

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

#### Implicit Conformance

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

Because modelling frameworks can only load models that conform to their metamodel, user-driven migration is always a manual process, in which models are migrated without using the modelling framework. Typically then, the metamodel user can only perform migration by editing the model directly, normally manipulating its underlying representation (e.g. XMI). Model editors and model management operations, which are ordinarily integral to MDE, cannot be used to manage models that do not conform to their metamodel and hence, cannot be used during model migration.

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

### 4.2.2 User-Driven Co-Evolution

Examples of co-evolution were analysed to discover and compare existing techniques for managing co-evolution. As discussed above, the separation of models and metamodels leads to two processes for co-evolution: *developer-driven* and *user-driven*. Analysis of the co-evolution examples identified in Section 4.1 highlighted several instances of user-driven co-evolution. Projects conducted in collaboration with Barber and with Sampson involved user-driven co-evolution, and two of the co-evolution examples taken from the xText project were managed in a user-driven manner. This section demonstrates user-driven co-evolution using a scenario similar to one observed during the collaboration with Barber.

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

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

### Scenario

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

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

Employing user-driven migration, Heather must trace and repair the loading error directly in the model as it is stored on disk. Model storage formats have typically been optimised to either reduce the size of models on disk or to improve the speed of random access to model elements. Therefore, human usability is not a key requirement for model storage formats. XMI [OMG 2007c], for example, is a standard model storage format and is regarded as sub-optimal for use by humans [OMG 2004]. Consequently, using a model storage format to perform model migration can be error-prone and tedious. When directly editing the underlying format of a model, reconciling conformance is often a slow and iterative process. For example, EMF [Steinberg *et al.* 2008], arguably the most widely used modelling framework, uses a multi-pass model parser and hence only reports one category of errors when a model cannot be loaded. After fixing one set of errors, another may be reported. In some cases, models are stored in a binary format and must be changed using a specialised editor, further impeding user-driven co-evolution. For example, models stored using the Connected Data Objects Model Repository (CDO) [Eclipse 2010] are persisted in a relational database, which must be manipulated when non-conformant models are to be edited.

#### Challenges

The above scenario highlights the two most significant challenges faced when performing user-driven co-evolution. Firstly, the underlying model representation is unlikely to be optimised for human usability and hence user-driven co-evolution is error-prone and tedious. Secondly, although conformance can be affected when a new version of a metamodel is installed, conformance problems are not reported to the user as part of the installation process. These challenges are further elaborated in the Section 4.3, which identifies research requirements.

It is worth noting that the above scenario describes a metamodel with only one user. Some metamodels – such as UML, Ecore, and MOF – have many more users, and user-driven co-evolution would require repeated manual effort from each user. In spite of this, UML, for example, does not provide a strategy for migrating between versions of the specification, and users must infer the migration semantics from changes to the specification.

### 4.2.3 Developer-Driven Co-Evolution Approaches

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

As noted in Section 4.2.2, existing co-evolution research focuses on developer-driven rather than user-driven co-evolution. By applying existing co-evolution approaches to the co-evolution examples identified in Section 4.1, existing developer-driven co-evolution approaches were categorised, compared and contrasted. Appendix A describes the way in which the co-evolution examples were explored with several existing developer-driven co-evolution approaches. From the analysis, three categories of developer-driven co-evolution approach were identified: *manual specification*, *operator-based* and *inference*. This categorisation has been published in [Rose *et al.* 2009b]. Each category is now discussed.

#### Manual Specification

In *manual specification*, the migration strategy is encoded manually by the metamodel developer, typically using a general purpose programming language (e.g. Java) or a model-to-model transformation language (such as QVT [OMG 2005], or ATL [Jouault & Kurtev 2005]). The migration strategy can manipulate instances of the metamodel in any way permitted by the modelling framework. Manual specification approaches have been used to manage migration in the Eclipse GMF project [Gronback 2009] and the Eclipse MDT UML2 project [Eclipse 2009b]. Compared operator-based and inference techniques (below), manual specification permits the metamodel developer the most control over model migration.

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

### Operator-based

In *operator-based co-evolution* techniques, a library of *co-evolutionary operators* is provided. Each co-evolutionary operator specifies a metamodel evolution along with a corresponding model migration strategy. For example, the "Make Reference Containment" operator might evolve the metamodel such that a non-containment reference becomes a containment reference and migrate models such that the values of the evolved reference are replaced by copies. By composing co-evolutionary operators, metamodel evolution can be performed and a migration strategy can be generated without writing any code. [Wachsmuth 2007] proposes a library of co-evolutionary operators for MOF metamodels. COPE [Herrmannsdoerfer *et al.* 2009b] is an operator-based co-evolution approach for the Eclipse Modeling Framework (EMF) [Steinberg *et al.* 2008].

The efficacy of an operator-based co-evolution approach depends heavily on the richness of its library of co-evolutionary operators. When no operator describes the required co-evolution pattern, the metamodel developer must use another approach for performing model migration. For instance, COPE allows migration to be specified manually with a general purpose programming language when no co-evolutionary operator is appropriate. (Consequently, custom migration strategies in COPE suffer one of the same limitations as manual specification approaches: model migration patterns are not captured in the language used to specify migration strategies).

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

Clear communication of the effects of each co-evolutionary operator (on both the metamodel and its instance models) can improve the navigability of large libraries of co-evolutionary operators. COPE, for example, provides a name, description, list of parameters and applicability constraints for each co-evolutionary operator. An example, taken from COPE's library[5], is shown below. To choose between operators, users can read descriptions (such as the one shown below) examine the source code of the operator, or try executing the operator (an undo command is provided).

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

Finding a balance between richness and navigability is a key challenge in defining libraries of co-evolutionary operators for operation-based co-evolution approaches. Analogously, a known challenge in the design of software interfaces is the trade-off between a rich and a concise interface [Bloch 2005].

To perform metamodel evolution using co-evolutionary operators, the library of co-evolutionary operators must be integrated with tools for editing metamodels. COPE, for instance, provides integration with the EMF tree-based metamodel editor. However, some developers edit their metamodels using a textual syntax, such as Emfatic [IBM 2005]. In general, freeform text editing is less restrictive than tree-based editing (because in the latter, the

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

metamodel is always structurally sound whereas in the former, the text does not always have to compile). Consequently, it is not clear whether operator-based co-evolution can be used with all categories of metamodel editing tool.

### Inference

In *inference* approaches, a migration strategy is derived from the evolved metamodel and the *metamodel history*. Inference approaches can be further categorised according to the type of metamodel history used. *Differencing* approaches compare and match the original and evolved metamodels, while *change recording* approaches use a record of primitive changes made to the original metamodel to produce the evolved metamodel. The analysis of the evolved metamodel and the metamodel history yields a *difference model* [Cicchetti *et al.* 2008], a representation of the changes between original and evolved metamodel. The difference model is used to infer a migration strategy, typically by using a higher-order model-to-model transformation[6] to produce a model-to-model transformation from the difference model. [Cicchetti *et al.* 2008] and [Garcés *et al.* 2009] describe differencing-based inference approaches. There exist no pure change recording approaches, although COPE [Herrmannsdoerfer *et al.* 2009b] uses change recording to support the specification of custom model migration strategies, and [Méndez *et al.* 2010] suggest that change recording approach might be used to manage metamodel-transformation co-evolution.

Compared to manual specification and operator-based co-evolution approaches, inference approaches require the least amount of effort from the metamodel developer who needs only to evolve the metamodel and provide a metamodel history. However, for some types of metamodel change, there is more than one feasible model migration strategy. For example, when a metaclass is deleted, one feasible migration strategy is to delete all instances of the deleted metaclass. Alternatively, the type of each instance of the deleted metaclass could be changed to another metaclass that specifies equivalent structural features.

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

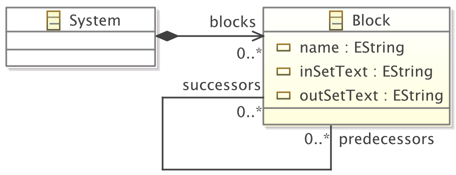
Because inference approaches use heuristics to select a migration strategy, it can sometimes be difficult to reason about which migration strategy will be selected. For domains where predictability, completeness and correctness are a primary concern (e.g. safety critical or security critical systems,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

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

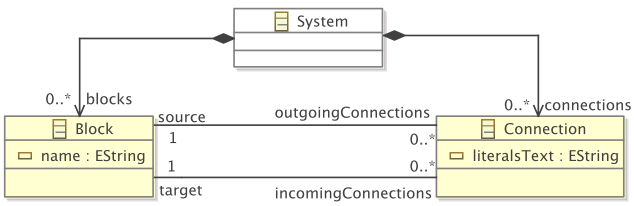
systems that must undergo certification with respect to a relevant standard), such approaches are unsuitable, and deterministic approaches that can be demonstrated to produce correct, predictable results will be required.

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

**Example** The following example was observed during the development of the Epsilon FPTC tool (summarised in Section 4.1 and described in [Paige *et al.* 2009]). The source code is available from EpsilonLabs[7]. Figure 4.3(a) illustrates the original metamodel in which a `System` comprises any number of `Blocks`. A `Block` has a name, and any number of `successor` `Blocks`; `predecessors` is the inverse of the `successors` reference.




(a) Original metamodel, prior to evolution




(b) Evolved metamodel with Connection metaclass

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

Further analysis of the domain revealed that extra information about the relationship between `Blocks` was to be captured. The evolved metamodel is shown in Figure 4.3(b). The `Connection` class is introduced to capture the extra information via its `literalsText` attribute. `Blocks` are no longer related directly to `Blocks`, instead they are related via an instance of the `Connection` class. The `incomingConnections` and `outgoingConnections` references of `Block` are used to relate `Blocks` to each other via an instance of `Connection`.

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

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

the strategy used by the Epsilon FPTC tool to migrate a model from original to evolved metamodel and is taken from [Rose *et al.* 2009b]:

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

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

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

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

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

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

2. And nothing else changes.

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

**Change recording** In change recording approaches, metamodel evolution is monitored by a tool, which records a list of primitive changes (e.g. Add class named `Connection`, Change the type of feature `successors` from `Block` to `Connection`). The record of changes may be reduced to a normal form to remove redundancy, but doing so can erase useful information. In change recording, some types of metamodel evolution can be more easily recognised than with differencing. With change recording, renaming can be distinguished from a deletion followed by an addition. With differencing, this distinction is not possible.

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

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

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

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

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

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

### 4.2.4 Summary

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

Modern MDE modelling frameworks separate models and metamodels and, hence, co-evolution is a two-step process. To facilitate model migration, metamodel developers may codify an executable migration strategy and distribute it along with the evolved metamodel (developer-driven co-evolution). When no executable migration strategy is provided, models must be migrated by hand (user-driven co-evolution). Because modelling frameworks implicitly enforce conformance, user-driven co-evolution is performed by editing the underlying storage representation of models, which is error-prone and tedious.

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

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

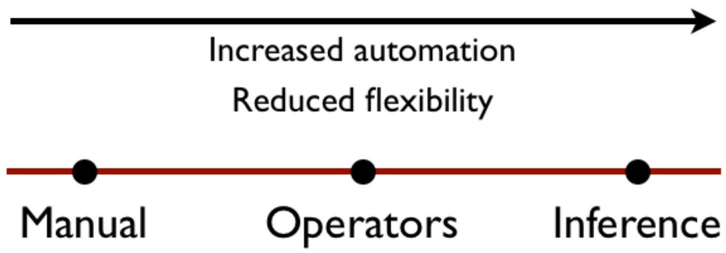



Figure 4.4: Spectrum of developer-driven co-evolution approaches

Manual specification affords the metamodel developer more flexibility in the specification of the migration strategy, but, because languages that do not capture re-occurring model migration patterns are typically used, may require more effort. By contrast, inference approaches derive a migration strategy from a metamodel history and hence require less effort from the metamodel developer. However, an inference approach affords the metamodel developer less flexibility, and may restrict the metamodel evolution process because, for example, the order of metamodel changes affects the inference of a migration strategy. Operator-based approaches occupy the middle-ground: by restricting the way in which metamodel evolution is expressed, an operator-based approach can be used to infer a migration strategy. The metamodel developer selects appropriate operators that express both metamodel evolution and model migration. Operator-based approaches require a specialised metamodel editor, and it is not yet clear whether they can be applied when a metamodel is represented with a freeform (e.g. textual) rather than a structured (e.g. tree-based) syntax.

## 4.3 Requirements Identification

The analysis presented throughout this chapter has highlighted a number of challenges for identifying and managing model-metamodel co-evolution. Several factors affect and restrict the way in which co-evolution is performed in practice. The way in which modelling frameworks are implemented affect the ways in which the impact analysis and propagation of metamodel changes can be performed. Existing co-evolution approaches are developer-driven rather than user-driven (i.e. assume that the metamodel developer will provide a migration strategy), which was not the case for several of the examples identified in Section 4.1. Additionally, the languages used to specify model migration vary over existing co-evolution approaches, which inhibits the conceptual and practical re-use of model migration patterns. This section contributes requirements for structures and processes that seek to address these challenges.

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

### 4.3.1 Explicit conformance checking

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

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

### 4.3.2 User-driven co-evolution

When a metamodel change will affect conformance in only a small number of models, a metamodel developer may decide that the extra effort required to specify an executable migration strategy is too great, and prefer a user-driven co-evolution technique. Section 4.2.2 introduced – and highlighted several challenges for – user-driven co-evolution.

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

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

### 4.3.3 Developer-driven co-evolution

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

Of the three categories of developer-driven co-evolution technique identified in Section 4.2.3, only manual specification (in which the metamodel developer specifies the migration strategy by hand) always requires the use of a migration strategy language. Nevertheless, both operator-based and inference approaches might utilise a migration strategy language in particular circumstances. Some operator-based approaches, such as COPE, permit manual specification of a model migration strategy when no co-evolutionary operator is appropriate. For describing the effects of co-evolutionary operators, the model migration part of an operator could be described using a model migration strategy language. When an inference approach suggests more than one feasible migration strategy, a migration strategy language could be used to present alternatives to the metamodel developer. To some extent then, the choice of model migration strategy language influences the efficacy of all categories of developer-driven co-evolutionapproach.

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

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

Devising a domain-specific languages or DSL (discussed in Chapter 3) is one mechanism for capturing re-occurring patterns. Executable DSLs are often used for performing model management in contemporary model-driven development environments. DSLs are provided by model-to-model (M2M) transformation tools such as ATL [ATLAS 2007], VIATRA [Varró & Balogh 2007], workflow architectures such as oAW [openArchitectureWare 2007], and model-to-text (M2T) transformation tools such as MOFScript [Oldevik *et al.* 2005] and XPand [openArchitectureWare 2007].

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

## 4.4 Chapter Summary

The literature review performed in Chapter 3 identified several types of evolution that occur in MDE projects, including model refactoring, synchronisation and co-evolution. Although several papers propose structures and processes for managing evolution in MDE, little work has considered the way in which MDE artefacts evolve in practice. The work described in this chapter has investigated evolution in existing MDE projects, culminating in a deeper understanding of the conceptual and technical issues faced when identifying and managing co-evolution. Furthermore, the analysis has facilitated the derivation of requirements for structures and processes that will address several of the challenges to identifying and managing co-evolution today.

Examples of co-evolution were identified from real-world MDE projects, and supplementary data was located by examining a related area (refactoring in object-oriented systems) and from collaborative work on two projects using MDE. The examples were used to understand how co-evolution is performed in practice, and led to the identification of user-driven co-evolution. Furthermore, the examples were used to analyse and categorise existing approaches to managing co-evolution.

Examining the co-evolution examples and applying existing co-evolution tools to the examples led to several observations. Firstly, modelling frameworks restrict the way in which co-evolution can be identified and managed. Secondly, user-driven co-evolution (in which models are migrated without an executable strategy) occurs in practice, but no existing co-evolution tools provide support for it. Finally, the variation of languages used for specifying model migration inhibits the re-use of commonly occurring patterns.

From the analysis performed in this chapter, requirements for the implementation phase of the thesis were formulated. The structures and process developed to approach those requirements are described in Chapter 5, and seek to alleviate the restrictions of modelling frameworks, to improve and support user-driven co-evolution (which is currently error-prone and tedious), and to provide a common language for specifying model migration.

# Chapter 5

# Implementation

Section 4.3 presented requirements for structures and processes for identifying and managing co-evolution. This chapter describes the way in which the requirements have been approached. Several related structures were implemented, using domain-specific languages, metamodelling and model management operations. Figure 5.1 summarises the contents of the chapter. To facilitate the management of non-conformant models with existing modelling frameworks, a metamodel-independent syntax was devised and implemented (Section 5.1). To address some of the challenges faced in user-driven co-evolution, an OMG specification for a textual modelling notation was implemented (Section 5.2). To determine their merits and drawbacks, existing languages for specifying model migration were identified, analysed and compared (Section 5.3). Finally, a new model transformation language – tailored for model migration and centred around a novel approach to relating source and target model elements – was designed and implemented (Section 5.4).

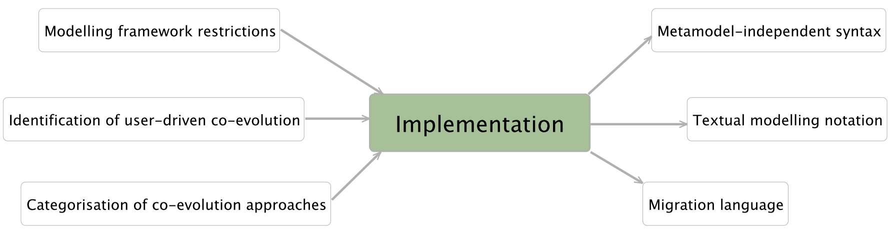



Figure 5.1: Implementation chapter overview.

## 5.1 Metamodel-Independent Syntax

Section 4.2.1 discussed the way in which modelling frameworks implicitly enforce conformance, and hence cannot be used to load non-conformant models.

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

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

### 5.1.1 Model Storage Representation

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

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

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

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

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

```
6 </xmi:XMI>
```

Listing 5.1: Exemplar person model in XMI

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

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

### 5.1.2 Binding to a generic metamodel

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

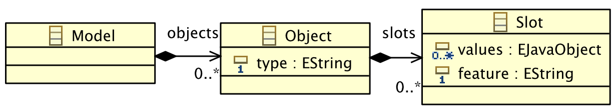



Figure 5.2: A generic metamodel.

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

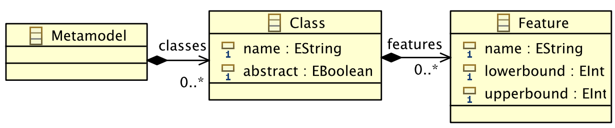



Figure 5.3: Minimal MOF metamodel.

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

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

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

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

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

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

The way in which type-compatibility is checked depends on the way in which the modelling framework is implemented, and on its underlying programming language. The Eclipse Modeling Framework (EMF) [Steinberg *et al.* 2008], for example, is implemented in a strongly-typed language (Java) and exposes some services for checking the type compatibility of model data with metamodel features. Metamodel features are typed. EMF provide methods for determining the underlying programming language representation, which can be used to implement type compatibility checks.

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

### 5.1.3 Example

By binding a model not to the underlying programming languages types defined in its metamodel but to the generic metamodel presented in Figure 5.2, conformance can be checked using the above constraints. Binding the exemplar XMI in Listing 5.1 to the generic metamodel shown in Figure 5.2 produces three instances of Object. A UML object diagram for this instantiation of the generic metamodel is shown in Figure 5.4. Instances of Object are shaded, while instances of Slot are not.

Binding the XMI in Listing 5.1 to the generic metamodel yields three Objects, each containing a slot whose feature is "name". The value of each slot varies: one has the value "Franz", another the value "Julie" and yet another the value "Hermann". The object containing the slot with value

"Franz" contains two further slots: one whose feature is "mother" and whose value is a reference to the object that contains the name slot with the value "Julie"[1] and one whose feature is "father" and whose value is a reference to the object containing "Hermann".

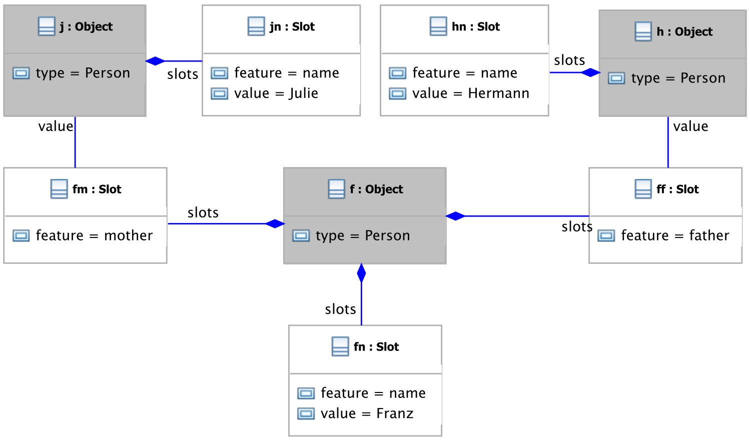



Figure 5.4: Exemplar instantiation of generic metamodel.

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

### 5.1.4 Applications

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

In Section 5.2, a textual modelling notation is integrated with the metamodel-independent syntax. In Section 5.4, a domain-specific language for migration

[1]Reference values in the generic metamodel are implemented using the proxy design pattern [Gamma *et al.* 1995].

uses metamodel independent syntax to perform partial migration by producing models that conform to a generic metamodel rather than their evolved metamodel.

One of the model migration tools discussed in Section 5.3, COPE [Herrmannsdoerfer *et al.* 2009] uses a metamodel-independent syntax. The strengths and weaknesses of using a metamodel-independent syntax in that context are described in Sections 5.3.2 and 5.3.3.

### Automatic Consistency Checking

In addition to the applications outlined above, a metamodel-independent syntax is potentially useful during metamodel installation. As discussed in Section 4.2.1, metamodel developers do not have access to downstream models, and conformance is implicitly enforced by modelling frameworks. Consequently, the conformance of models may be affected by the installation of a new version of a metamodel, and the conformance of models cannot be checked during installation. Typically, installing a new version of a metamodel can result in models that no longer conform to their metamodel and cannot be used with the modelling framework. Moreover, a user discovers conformance problems only when attempting to use a model after installation has completed, and not as part of the installation process.

To enable conformance checking as part of metamodel installation in EMF, the metamodel-independent syntax has been integrated with Concordance [Rose *et al.* 2010c] in joint work with Dimitrios Kolovos, a lecturer in this department, Nicholas Drivalos, a Research Associate in this department and James Williams, a research student in this department. Concordance provides a mechanism for resolving inter-model references (such as those between models and their metamodels), and can be used to efficiently determine the instances of a metamodel. Without Concordance, determining the the instances of a metamodel is possible only by checking every model in the workspace.

Integrating Concordance and the metamodel-independent syntax resulted in a service, executed after the installation of a metamodel, which identifies the models that are affected by the metamodel changes. All models that conform to the old version of the metamodel are checked for conformance with the new metamodel. As such, conformance checking occurs automatically and immediately after metamodel installation. Conformance problems are detected and reported immediately, rather than when the user next attempts to load an affected model.

### Summary

Modelling frameworks implicitly enforce conformance, which presents challenges for managing co-evolution. In particular, detecting and reconciling conformance problems involves managing non-conformant models, which cannot be loaded by modelling frameworks and hence cannot be used with model editors or model management operations. The metamodel-independent syntax proposed in this section enables modelling frameworks to load non-conformant models by binding models to a generic metamodel. The metamodel-independent syntax has been integrated with Concordance [Rose *et al.* 2010c] to facilitate the reporting of conformance problems during metamodel installation, and underpins the implementation of the textual modelling notation presented in Sections 5.2. The benefits and drawbacks of the metamodel-independent syntax in the context of user-driven co-evolution are explored in Chapter 6.

## 5.2 Textual Modelling Notation

The analysis of co-evolution examples in Chapter 4 highlighted two ways in which co-evolution is managed. In *developer-driven* co-evolution, migration is specified by the metamodel developer in an executable format; while in *user-driven co-evolution* migration is specified by the metamodel developer in prose or not at all. Performing user-driven co-evolution with modelling frameworks presents two key challenges that have not been explored by existing research. Firstly, user-driven co-evolution frequently involves editing the storage representation of the model, such as XMI. Model storage representations are typically not optimised for human use and hence user-driven co-evolution can be error-prone. Secondly, non-conformant model elements must be identified during user-driven co-evolution. When a multi-pass parser is used to load models, as is the case with EMF, not all conformance problems are reported at once, and user-driven co-evolution is an iterative process. In Section 4.3, these challenges lead to the identification of the following requirement: *This thesis must demonstrate a user-driven co-evolution process that enables the editing of non-conformant models without directly manipulating the underlying storage representation and provides a sound and complete conformance report for the original model and evolved metamodel.*

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

### 5.2.1 Human-Usable Textual Notation

The OMG's Human-Usable Textual Notation (HUTN) [OMG 2004] defines a textual modelling notation, which aims to conform to human-usability criteria [OMG 2004]. There is no current reference implementation of HUTN: the Distributed Systems Technology Centre's TokTok project (an implementation of the HUTN specification) is inactive (and the source code can no longer be found), whilst implementation of HUTN described in [Muller & Hassenforder 2005] has been abandoned in favour of Sintaks[2], which operates on domain-specific concrete syntax.

Model storage representations are often optimised for reducing storage space or increasing the speed of random access, rather than for human usability. By contrast, the HUTN specification states its primary design goal as human-usability and "this is achieved through consideration of the successes and failures of common programming languages" [OMG 2004, Section 2.2]. The HUTN specification refers to two studies of programming language usability to justify design decisions, but, because no reference implementation exists, the specification does not evaluate the human-usability of the notation. As HUTN is optimised for human-usability, using HUTN rather than XMI for user-driven co-evolution might lead to increased developer productivity. This claim is explored in Chapter 6.

Like the generic metamodel presented in Section 5.1, HUTN is a metamodel-independent syntax for MOF. However, the HUTN specification focuses on concrete syntax, whereas the metamodel-independent syntax presented in Section 5.1 focuses on abstract syntax. In this section, the key features of HUTN are introduced, and the sequel introduces a reference implementation of HUTN. To illustrate the notation, the MOF-based metamodel of families in Figure 5.5 is used. The `nuclear` attribute on the `Family` class is used to indicate that the family "comprises only a father, a mother, and children." [Merriam-Webster 2010].

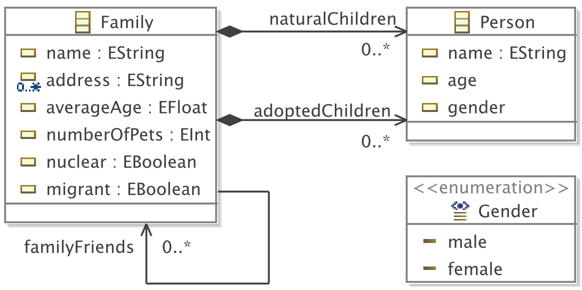



Figure 5.5: Exemplar families metamodel

## Basic Notation

Listing 5.2 shows the construction of an *object* in HUTN, here an instance of the Family class from Figure 5.5. Line 1 specifies the package containing the classes to be constructed (`FamilyPackage`) and a corresponding identi-

[2]`http://www.kermeta.org/sintaks/`

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

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

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

Listing 5.2: Specifying attributes with HUTN.

```
FamilyPackage "families" {
   Family "The Smiths" {
      naturalChildren: Person "John" { name: "John" },
                       Person "Jo" { gender: female }
   }
}
```

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

```
FamilyPackage "families" {
   Family "The Smiths" {
      familyFriends: Family "The Does"
   }
   Family "The Does" {}
}
```

Listing 5.4: Specifying a simple reference with HUTN.

### Keywords and Adjectives

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

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

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

Listing 5.5: Using keywords and adjectives in HUTN.

### Inter-Package References

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

```
FamilyPackage "families" {
   Family "The Smiths" {}
}
VehiclePackage "vehicles" {
   Vehicle "The Smiths' Car" {
      owner: FamilyPackage.Family "families"."The Smiths"
   }
}
```

Listing 5.6: Referencing objects in other packages with HUTN.

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

The HUTN specification defines name scope optimisation rules, which allow the definition above to be simplified to: `owner: Family "The Smiths"`, assuming that the VehiclePackage does not define a Family class, and that the identifier “The Smiths” is not used in the VehiclePackage block.

### Alternative Reference Syntax

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

```
FamilyPackage "families" {
   Family "The Smiths" {}
   Family "The Does" {}
   Family "The Bloggs" {}

   familyFriends {
      "The Does" "The Smiths"
      "The Does" "The Bloggs"
   }
}
```

Listing 5.7: Using a reference block in HUTN.

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

```
FamilyPackage "families" {
   Family "The Smiths" {}
   Family "The Does" {}
   Family "The Bloggs" {}

   Family "The Smiths" familyFriends Family "The Does";
   Family "The Smiths" familyFriends Family "The Bloggs";
}
```

Listing 5.8: Using an infix reference in HUTN.

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

### Customisation via Configuration

Some limited customisation of HUTN for particular metamodels can be achieved using *configuration files*. Customisations permitted include a parametric form of object instantiation; renaming of metamodel elements; specifying the default value of a feature; and providing a default identifier for classes of object.

### 5.2.2 Epsilon HUTN

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

#### Implementation of Epsilon HUTN

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

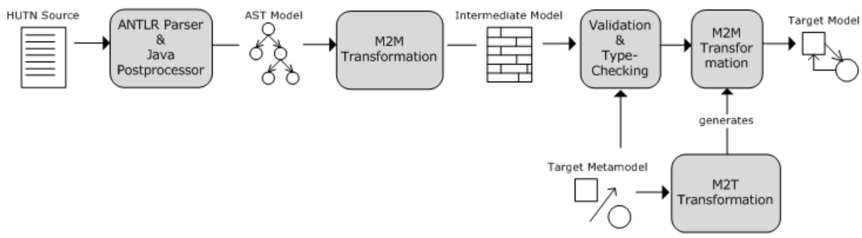



Figure 5.6: The architecture of Epsilon HUTN.

Figure 5.6 outlines the workflow through Epsilon HUTN, from HUTN source text to instantiated target model. The HUTN model specification is parsed to an abstract syntax tree using a HUTN parser specified in ANTLR [Parr 2007]. From this, a Java postprocessor is used to construct an instance of a simple AST metamodel (which comprises two meta-classes, Tree and Node).

Using ETL, M2M transformations are then applied to produce an intermediate model, which is an instance of the generic metamodel discussed in Section 5.1. Finally, a M2T transformation on the target metamodel, specified in EGL, produces a further M2M transformation, which consumes the intermediate model and produces the target model.

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

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

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

**AST Model to Intermediate Model** Epsilon HUTN uses ETL [Kolovos *et al.* 2008a] for specifying M2M transformation. One of the transformation rules from Epsilon HUTN is shown in Listing 5.9. The rule transforms a name node in the AST model (which could represent a package or a class object) to a package object in the intermediate model. The guard (line 5) specifies that a name node will only be transformed to a package object if the node has no parent (i.e. it is a top-level node, and hence a package rather than a class). The body of the rule states that the type, line number and column number of the package are determined from the text, line and column attributes of the node object. On line 11, a containment slot is instantiated to hold the children of this package object. The children of the node object are transformed to the intermediate model (using a built-in method, `equivalent()`), and added to the containment slot.

```
1 rule NameNode2PackageObject
2    transform n : AntlrAst!NameNode
```

```
   to p : Intermediate!PackageObject {

   guard : n.parent.isUndefined()

   p.type := n.text;
   p.line := n.line;
   p.col := n.column;

   var slot := new Intermediate!ContainmentSlot;
   for (child in n.children) {
      slot.objects.add(child.equivalent());
   }
   if (slot.objects.notEmpty()) {
      p.slots.add(slot);
   }
}
```

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

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

```
context ClassObject {
   constraint IdentifiersMustBeUnique {
      guard: self.id.isDefined()
      check: ClassObject.all
               .select(c|c.id = self.id).size() = 1;
      message: 'Duplicate identifier: ' + self.id
   }
}
```

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

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

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

**Intermediate Model to Target Model** When the intermediate model conforms to the target metamodel, the intermediate model can be transformed to an instance of the target metamodel. In generating the target model from the intermediate model (Figure 5.6), the transformation uses information from the target metamodel, such as the names of classes and features. A typical approach to this category of problem is to use a higher-order transformation on the target metamodel to generate the desired transformation. Epsilon HUTN uses a different approach: the transformation to the target model is produced by executing a M2T transformation on the target metamodel. EGL [Rose *et al.* 2008b] is a template-based M2T language. `[% %]` tag pairs are used to denote dynamic sections, which may produce text when executed. Any code not enclosed in a `[% %]` tag pair is included verbatim in the generated text.

Listing 5.11 shows part of the M2T transformation used by HUTN. The M2T transformation generates a M2M transformation which specifies the way in which the intermediate model is transformed to the target model. The loop beginning on line 1 of Listing 5.11 iterates over each meta-class in the target metamodel, producing a M2M transformation rule. The generated transformation rule consumes a class objects in the intermediate model and produces an element of the target model. The guard of the generated transformation rule (line 6) ensures that only class objects with a type equal to the current meta-class are transformed by the generated rule. To generate the body of the rule, the M2T transformation iterates over each structural feature of the current meta-class, and generates appropriate transformation code for populating the values of each structural feature from the slots on the class object in the intermediate model. The part of the M2T transformation that generates the body of M2M transformation rule is omitted in Listing 5.11 because it contains a large amount of code for interacting with EMF, which is not relevant to this discussion.

```
1  [% for (class in EClass.allInstances()) { %]
2  rule Object2[%=class.name%]
3   transform o : Intermediate!ClassObject
4   to t : Model![%=class.name%] {
5
6     guard: o.type = '[%=class.name%]'
7
8     -- body omitted
9   }
10 [% } %]
```

Listing 5.11: Part of the M2T transformation (in EGL) for generating the intermediate model to target model transformation (in ETL).

For example, executing the M2T transformation in Listing 5.11 on the Families metamodel (Figure 5.5) generates the two M2M transformation rules shown in Listing 5.12. The rules produce instances of Family and Person from instances of ClassObject in the intermediate model. The body of each rule copies the values from the slots of the ClassObject to the Family or Person in the target model. Lines 7-9, for example, copy the value of the name slot (if one is specified) to the target Family.

```
1  rule Object2Family
2   transform o : Intermediate!ClassObject
3   to t : Model!Family {
4
5     guard: o.type = 'Family'
6
7     if (o.hasSlot('name')) {
8       t.name := o.findSlot('name').values.first;
9     }
10
11    if (o.hasSlot('address')) {
12      for (value in o.findSlot('address').values) {
13        t.address.add(value);
14      }
15    }
16
17    -- remainder of body omitted
18  }
19
20 rule Object2Person
21  transform o : Intermediate!ClassObject
22  to t : Model!Person {
23
24    guard: o.type = 'Person'
25
26    if (o.hasSlot('name')) {
27      t.name := o.findSlot('name').values.first;
28    }
29
30    -- remainder of body omitted
31  }
```

Listing 5.12: The M2M transformation generated by HUTN for the Families metamodel

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

### 5.2.3 Migration with HUTN

Migrating non-conformant models with HUTN involves transforming the XMI of the non-conformant model to HUTN, reconciling the conformance problems in the HUTN source by hand, and transforming the reconciled HUTN back to XMI. The transformations to and from XMI can be automated, and the way in which Epsilon HUTN transforms from HUTN to XMI was described above. Because Epsilon HUTN uses the metamodel-independent syntax (from Section 5.1) as an intermediary, transformation from XMI to HUTN is implemented as follows: the metamodel-independent syntax is used to parse the XMI and produce an intermediate model. Subsequently, an unparser (implemented using the visitor design pattern [Gamma *et al.* 1995]) generates HUTN source for each element of the intermediate model. In this manner, HUTN can be generated for any XMI document, regardless of whether the model described by the XMI conforms to its metamodel.

To demonstrate the way in which HUTN can be used to perform migration, the exemplar XMI shown in Listing 5.1 is represented using HUTN in Listing 5.13. Recall that the XMI describes three Persons, Franz, Julie and Hermann. Julie and Hermann are the mother and father of Franz.

```
Persons "kafkas" {
   Person "Franz" { name: "Franz" }
   Person "Julie" { name: "Julie" }
   Person "Hermann" { name: "Hermann" }

   Person "Franz" mother Person "Julie";
   Person "Franz" father Person "Hermann";
}
```

Listing 5.13: HUTN for people with mothers and fathers.

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

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

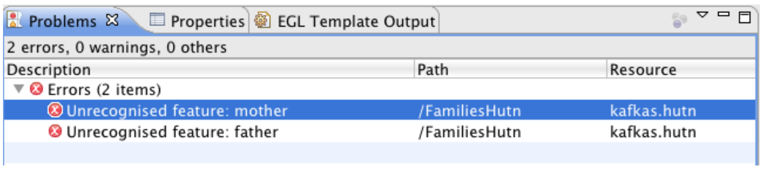



Figure 5.7: Conformance problem reporting in Epsilon HUTN.

Resolving the conformance problems requires the user to change the feature named in the infix associations from mother (father) to parents. The Epsilon

HUTN development tools provide content assistance, which might be useful in this situation. Listing 5.14 shows a HUTN document that conforms to the metamodel defining parents rather than mother and father.

```
Persons "kafkas" {
   Person "Franz" { name: "Franz" }
   Person "Julie" { name: "Julie" }
   Person "Hermann" { name: "Hermann" }

   Person "Franz" parents Person "Julie";
   Person "Franz" parents Person "Hermann";
}
```

Listing 5.14: HUTN for people with parents.

### 5.2.4 Limitations

During the development of Epsilon HUTN, two primary limitations of the notation became apparent. The first relates to the nature of a metamodel-independent syntax, and the latter to the suitability of HUTN for performing user-driven co-evolution.

#### Generic vs specific concrete syntax

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

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

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

The above expression is written using a domain-specific syntax. In HUTN, the specification of these behaviours is less concise. For example, Listing 5.15 gives the HUTN syntax for failure behaviour (5.1), above.

```
Behaviour {
   lhs: Tuple {
      contents: IdentifierSet { contents: Wildcard {} },
                IdentifierSet { contents: Wildcard {} }
```

```
  }

  rhs: Tuple {
    contents: IdentifierSet { contents: Fault "late" {} }
  }
}
```

Listing 5.15: Failure behaviour specified in HUTN.

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

In general, HUTN is less concise than a domain-specific syntax for metamodels containing a large number of classes with few attributes, and in cases where most attributes are used to define structural relationships among concepts. However, a domain-specific syntax is specified using metamodel concepts and hence can be affected by metamodel evolution.

#### Optimised XMI omits type information

When HUTN is used for user-driven co-evolution, non-conformant models (represented in XMI) are transformed to HUTN source. The default EMF serialisation mechanism is configure to reduce the size of models on disk and consequently the XMI produced by EMF omits type information when it may be inferred from a metamodel. This is problematic for managing co-evolution because, when a metamodel evolves, type information might be erased. The implementation of Epsilon HUTN has been extended to account for optimised XMI, and reports type inference errors along with conformance problems.

### 5.2.5 Summary

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

## 5.3 Analysis of Languages used for Migration

In contrast to the previous two sections, this section focuses not on *user-driven* but rather on *developer-driven* co-evolution, in which migration is specified in a programming language. Section 4.2.3 discussed existing approaches to model migration, highlighting variation in the languages used for specifying migration strategies. In this section, migration strategy languages are compared, using the example of metamodel evolution given in Section 5.3.1. From this comparison, requirements for a domain-specific language for specifying and executing model migration strategies are derived (Section 5.3.3). The sequel describes an implementation of a model model migration language based on the analysis presented here. The work described in this section has been published in [Rose *et al.* 2010f].

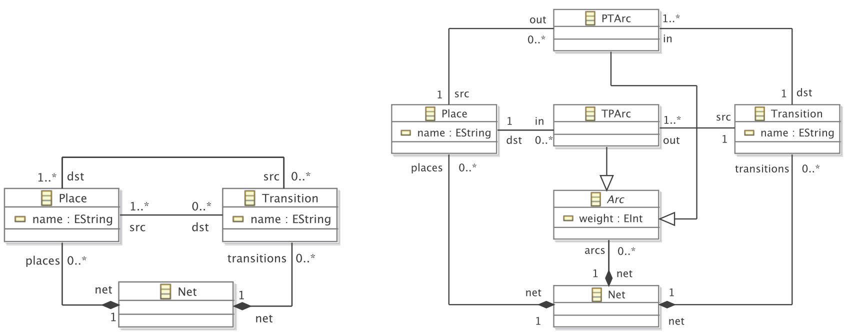
### 5.3.1 Co-Evolution Example

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

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

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

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




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

Figure 5.8: Exemplar metamodel evolution. Taken from [Rose *et al.* 2010f].

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

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

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

   Set s as the `src` of arc.

   Set `t` as the `dst` of arc.

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

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

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

   Set `t` as the `src` of arc.

   Set d as the `dst` of arc.

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

2. And nothing else changes.

### 5.3.2 Existing Approaches

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

#### Manual Specification with Model-to-Model Transformation

A model-to-model transformation specified between original and evolved metamodel can be used for performing model migration. Part of the model migration for the Petri nets metamodel is codified with the Atlas Transformation Language (ATL) [Jouault & Kurtev 2005] in Listing 5.16. Rules for migrating `Places` and `TPArcs` have been omitted for brevity, but are similar to the `Nets` and `PTArcs` rules.

Model transformation in ATL is specified using `rules`, which transform source model elements (specified using the `from` keyword) to target model elements (specified using `to` keyword). For example, the `Nets` rule on line 1 of Listing 5.16 transforms an instance of `Net` from the original (source) model to an instance of `Net` in the evolved (target) model. The source model element (the variable `o` in the `Net` rule) is used to populate the target model element (the variable `m`). ATL allows rules to be specified as *lazy* (not scheduled automatically and applied only when called by other rules).

The `Transitions` rule in Listing 5.16 codifies in ATL the migration strategy described previously. The rule is executed for each `Transition` in the original model, `o`, and constructs a `PTArc` (`TPArc`) for each reference to a `Place` in `o.src` (`o.dst`). Lazy rules must be used to produce the arcs to prevent circular dependencies with the `Transitions` and `Places` rules. Here, ATL, a typical rule-based transformation language, is considered and model migration would be similar in QVT. With Kermeta, migration would be specified in an imperative style using statements for copying `Nets`, `Places` and `Transitions`, and for creating `PTArcs` and `TPArcs`.

```
rule Nets {
 from o : Before!Net
 to m : After!Net ( places <- o.places, transitions <- o.transitions )
}

rule Transitions {
 from o : Before!Transition
 to m : After!Transition (
    name <- o.name,
    "in" <- o.src->collect(p | thisModule.PTArcs(p,o)),
    out <- o.dst->collect(p | thisModule.TPArcs(o,p))
   )
}

unique lazy rule PTArcs {
 from place : Before!Place, destination : Before!Transition
 to ptarcs : After!PTArc (
    src <- place, dst <- destination, net <- destination.net
   )
}
```

Listing 5.16: Fragment of the Petri nets model migration in ATL

In model transformation, [Czarnecki & Helsen 2006] identifies two common categories of relationship between source and target model, *new-target* and *existing-target*. In the former, the target model is constructed afresh by the execution of the transformation, while in the latter, the target model contains the same data as the source model before the transformation is executed. ATL supports both new- and existing-target relationships (the latter is termed a refinement transformation). However, ATL refinement transformations may only be used when the source and target metamodel are the same, as is typical for existing-target transformations.

In model migration, source and target metamodels differ, and hence existing-target transformations cannot be used to specify model migration strategies. Consequently, model migration strategies are specified with new-target model-to-model transformation languages, and often contain sections for copying from original to migrated model those model elements that have not been affected by metamodel evolution. For the Petri nets example, the `Nets` rule (in Listing 5.16) and the `Places` rule (not shown) exist only for this reason.

### Manual Specification with Ecore2Ecore Mapping

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

Because EMF binds models to their metamodel (discussed in Section 4.2.1), EMF cannot use an evolved metamodel to load an instance of the original metamodel. Therefore, Hussey and Paternostro's approach requires the metamodel developer to provide a mapping between the metamodelling language of EMF, Ecore, and the concrete syntax used to persist models, XMI. Mappings are specified using a tool that can suggest relationships between source and target metamodel elements by comparing names and types. For the Petri nets example, the mappings shown in Figure 5.9 were defined between the original and evolved metamodels.

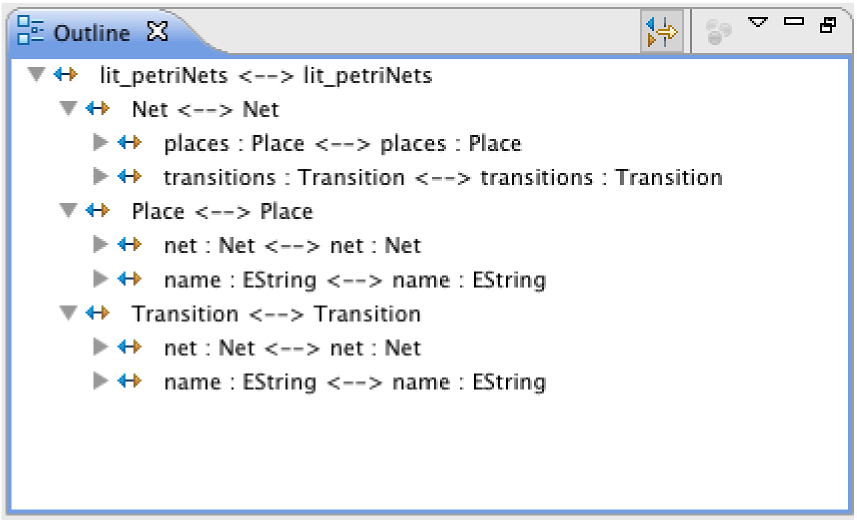



Figure 5.9: Mappings between the original and evolved Petri nets metamodels

The mappings are used by the EMF XMI parser to determine the metamodel types to which pieces of the XMI will be bound. When a type or feature is not bound, the user must specify a custom migration strategy in Java. For the Petri nets metamodel, the `src` and `dst` features of `Place` and `Transition` are not bound, because migration is more complicated than a one-to-one mapping. Instead, the migration of the `src` and `dst` features is specified with Java.

Model migration is specified on the XMI representation of the model and hence presumes some knowledge of the XMI standard. For example, in XMI, references to other model elements are serialised as a space delimited collection of URI fragments [Steinberg *et al.* 2008]. Listing 5.17 shows a section of the Ecore2Ecore model migration for the Petri net example presented above. The method shown converts a `String` containing URI fragments to a `Collection` of `Places`. The method is used to access the `src` and `dst` features of `Transition`, which no longer exist in the evolved metamodel and hence are not loaded automatically by EMF. To specify the migration strategy for the Petri nets example, the metamodel developer must know the way in which the `src` and `dst` features are represented in XMI. The complete listing, not shown here, exceeds 200 lines of code.

```java
private Collection<Place> toCollectionOfPlaces
(String value, Resource resource) {

  final String[] uriFragments = value.split("␣");
  final Collection<Place> places = new LinkedList<Place>();

  for (String uriFragment : uriFragments) {
    final EObject eObject = resource.getEObject(uriFragment);
    final EClass place = PetriNetsPackage.eINSTANCE.getPlace();

    if (eObject == null || !place.isInstance(eObject))
      // throw an exception

    places.add((Place)eObject);
  }

  return places;
}
```

Listing 5.17: Java method for deserialising a reference.

### Operator-based Co-evolution with COPE

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

To perform metamodel evolution using an operator-based approach, the library of co-evolutionary operators must be integrated with tools for editing metamodels. COPE provides integration with the EMF tree-based metamodel

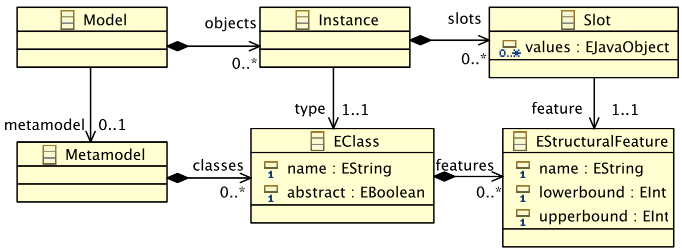



Figure 5.10: Simplification of the metamodel-independent representation used by COPE, based on [Herrmannsdoerfer *et al.* 2009b].

editor. Operators may be applied to an EMF metamodel, and COPE tracks their application. Once metamodel evolution is complete, a migration strategy can be generated automatically from the record of changes maintained by COPEs. The migration strategy is distributed along with the updated metamodel, and metamodel users choose when to execute the migration strategy on their models.

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

As discussed above, existing-target transformations cannot be used for specifying model migration strategies as the source (original) and target (evolved) metamodels differ. However, models can be structured independently of their metamodel using a metamodel-independent syntax, such as the one introduced in Section 5.1. Figure 5.10 shows a simplification of the metamodel-independent syntax used by COPE. By using a metamodel-independent syntax as an intermediary, an existing-target transformation can be used for performing model migration. Further details of this technique are given in [Herrmannsdoerfer *et al.* 2009b].

Listing 5.18 shows the COPE model migration strategy for the Petri net example given above[3]. Most notably, slots for features that no longer exist must be explicitly `unset`. In Listing 5.18, slots are `unset` on four occasions (on lines 2, 8 and 16), once for each feature that exists in the original metamodel but not the evolved metamodel. Namely, these features are: `src` and `dst` of `Transition` and of `Place`. Failing to `unset` slots that do not conform with the evolved metamodel causes migration to fail with an error.

[3]In Listing 5.18, some of the concrete syntax has been changed in the interest of readability.

```
for (transition in Transition.allInstances) {
  for (source in transition.unset('src')) {
    def arc = petrinets.PTArc.newInstance()
    arc.src = source; arc.dst = transition;
    arc.net = transition.net
  }

  for (destination in transition.unset('dst')) {
    def arc = petrinets.TPArc.newInstance()
    arc.src = transition; arc.dst = destination;
    arc.net = transition.net
  }
}

for (place in Place.allInstances) {
  place.unset('src'); place.unset('dst');
}
```

Listing 5.18: Petri nets model migration in COPE

### 5.3.3 Requirements Identification

By analysing the languages used for model migration in existing approaches to managing developer-driven co-evolution, requirements were derived for a domain-specific language for specifying and executing model migration. The derivation of the requirements is now summarised, by considering two orthogonal concerns: the source-target relationship of the language used for specifying migration strategies and the way in which models are represented during migration.

#### Source-Target Relationship

When migration is specified as a new-target transformation, as was the case for the ATL transformation shown in Listing 5.16, model elements that have not been affected by metamodel evolution must be explicitly copied from the original to the migrated model. When migration is specified as an existing-target transformation, as was the case for the COPE transformation shown in Listing 5.18, model elements and values that no longer conform to the target metamodel must be explicitly removed from the migrated model. By contrast, the Ecore2Ecore approach does not require explicit copying or unsetting code. Instead, the relationship between original and evolved metamodel elements is captured in a mapping model specified by the metamodel developer. The mapping model can be derived automatically and customised by the metamodel developer. To explore the appropriateness for model migration of an alternative to new- and existing-target transformations, the following requirement was derived: *The migration language must* ***automatically*** *copy every* *model element that conforms to the evolved metamodel from original to migrated model, and must not automatically copy any model element that does not conform to the evolved metamodel from original to migrated model.*

### Model Representation

When using the Ecore2Ecore approach, model elements that do not conform to the evolved metamodel are accessed by maniuplating XMI. Consequently, the metamodel developer must be familiar with XMI and must perform tasks such as dereferencing URI fragments (Listing 5.17) and type conversion. Transformation languages abstract away from the underlying storage representation of models (such as XMI) by using a modelling framework to load, store and access models. Consequently, migration strategies written in a transformation language need not manage details specific to XMI, such as dereferencing URI fragments. Furthermore, decoupling a transformation language from the model representation facilitates interoperability with more than one modelling technology, as demonstrated by the languages of the Epsilon platform [Kolovos 2009]. Consequently, the following requirement was identified: *The migration language must not expose the underlying representation of original or migrated models.*

To apply co-evolution operators, COPE requires the metamodel developer to use a specialised metamodel editor, which can manipulate only metamodels defined with EMF. Similarly, the mapping tool used in the Ecore2Ecore approach can be used only with metamodels defined with EMF. Although EMF is arguably the most widely-used modelling framework, other frameworks are used today. Adapting to interoperate with new systems is recognised as a common reason for software evolution [Sjøberg 1993], and as such migration between modelling frameworks should be regarded as a possible use case for a model migration language. To better support integration with modelling frameworks other than EMF, the following requirement was derived: *The migration language must be loosely coupled with modelling frameworks and must not assume that models and metamodels will be represented in EMF.*

## 5.4 Epsilon Flock: A Model Migration Language

Driven by the analysis presented above, a domain-specific language for model migration, Epsilon Flock (subsequently referred to as Flock), was designed and implemented. Section 5.4.1 discusses the principle tenets of Flock, which include user-defined migration rules and a novel algorithm for relating source and target model elements. In Section 5.4.2, Flock is demonstrated via application to three examples of model migration. The work described in this section has been published in [Rose *et al.* 2010f].

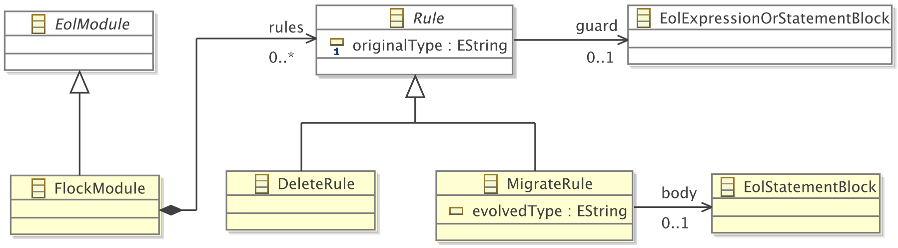



Figure 5.11: The abstract syntax of Flock.

### 5.4.1 Design and Implementation

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

Like Epsilon HUTN (Section 5.2.2), Flock is built atop Epsilon, which was described in Section 2.3.2. In particular, Flock uses the Epsilon Model Connectivity layer to provide interoperability with several modelling frameworks, and the Epsilon Object Language (EOL) for specifying the imperative part of user-defined migration rules.

#### Abstract Syntax

As illustrated by Figure 5.11, Flock migration strategies are organised into modules (`FlockModule`). Flock modules inherit from EOL modules (`EolModule`) and hence provide language constructs for specifying user-defined operations and for re-using modules. Flock modules comprise any number of rules (`Rule`). Each rule has an original metamodel type (`originalType`) and can optionally specify a `guard`, which is either an EOL statement or a block of EOL statements. `MigrateRules` must specify an evolved metamodel type (`evolvedType`) and/or a `body` comprising a block of EOL statements.

#### Concrete Syntax

Listing 5.19 shows the concrete syntax of migrate and delete rules. All rules begin with a keyword indicating their type (either `migrate` or `delete`), followed by the original metamodel type. Guards are specified using the `when`

```
migrate <originalType> (to <evolvedType>)?
(when (:<eolExpression>)|({<eolStatement>+}))? {
  <eolStatement>*
}

delete <originalType>
(when (:<eolExpression>)|({<eolStatement>+}))?
```

Listing 5.19: Concrete syntax of migrate and delete rules.

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

Note that Flock does not define a create rule. The creation of new model elements is instead encoded in the imperative part of a migrate rule specified on the containing type.

### Execution Semantics

When executed, a Flock module consumes an original model, O, and constructs a migrated model, M. The transformation is performed in three phases: rule selection, equivalence establishment and rule execution. The behaviour of each phase is described below, and the first example in Section 5.4.2 demonstrates the way in which a Flock module is executed.

**Rule Selection** The rule selection phase determines an *applicable* rule for every model element, e, in O. As such, the result of the rule selection phase is a set of pairs of the form <r,e> where r is a migration rule.

A rule, r, is *applicable* for a model element, e, when the original type of r is the same type as (or is a supertype of) the type of e; and the guard part of r is satisfied by e.

The rule selection phase has the following behaviour:

- For each original model element, e, in O:
  - Identify for e the set of all applicable rules, R. Order R by the occurrence of rules in the Flock source file.
    - If R is empty, let r be a default rule, which has the type of e as both its original and evolved type, and an empty body.
    - Otherwise, let r be the first element of R.
  - Add the pair <r,e> to the set of selected rules.

**Equivalence Establishment** The equivalence establishment phase creates an equivalent model element, e', in M for every pair of rules and original model elements, `<r,e>`. The equivalent establishment phase produces a set of triples of the form `<r,e,e′>`, and has the following behaviour:

- For each pair `<r,e>` produced by the rule selection phase:
  - If `r` is a delete rule, do nothing.
  - If `r` is a migrate rule:
    - Create a model element, `e′`, in M. The type of `e′` is determined from the the `evolvedType` (or the `originalType` when no `evolvedType` has been specified) of `r`.
    - Copy the data contained in `e` to `e′` (using the *conservative copy* algorithm described in the sequel).
    - Add the triple `<r,e,e′>` to the set of equivalences.

**Rule Execution** The final phase executes the imperative part of the user-defined migration rules on the set of triples `<r,e,e′>`, and has the following behaviour:

- For each triple `<r,e,e′>` produced by the equivalence establishment phase:
  - Bind `e` and `e′` to EOL variables named `original` and `migrated`, respectively.
  - Execute the body of `r` with EOL.

### Conservative Copy

Flock contributes an algorithm, termed *conservative copy*, that copies model elements from original to migrated model only when those model elements conform to the evolved metamodel. Conservative copy is a hybrid of the new- and existing-target source-target relationships that are commonly used in M2M transformation [Czarnecki & Helsen 2006].

Conservative copy operates on an original model element, `e`, and its equivalent model element in the migrated model, `e′`, and has the following behaviour:

- For each metafeature, `f` for which `e` has specified a value:
  - Find a metafeature, `f′`, of `e′` with the same name as `f`.
    - If no equivalent metafeature can be found, do nothing.
    - Otherwise, copy the original value (`e.f`) to produce a migrated value (`e′.f′`) if and only if the migrated value conforms to `f′`.

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

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

Epsilon includes a model connectivity layer (EMC), which provides a common interface for accessing and persisting models. Currently, EMC provides drivers for several modelling frameworks, permitting management of models defined with EMF, the Metadata Repository (MDR), Z or XML. To support migration between metamodels defined in heterogenous modelling frameworks, EMC was extended to include a conformance checking service, and each EMC driver to provide conformance checking semantics specific to its modelling framework. Flock implements conservative copy by delegating conformance checking responsibilities to EMC.

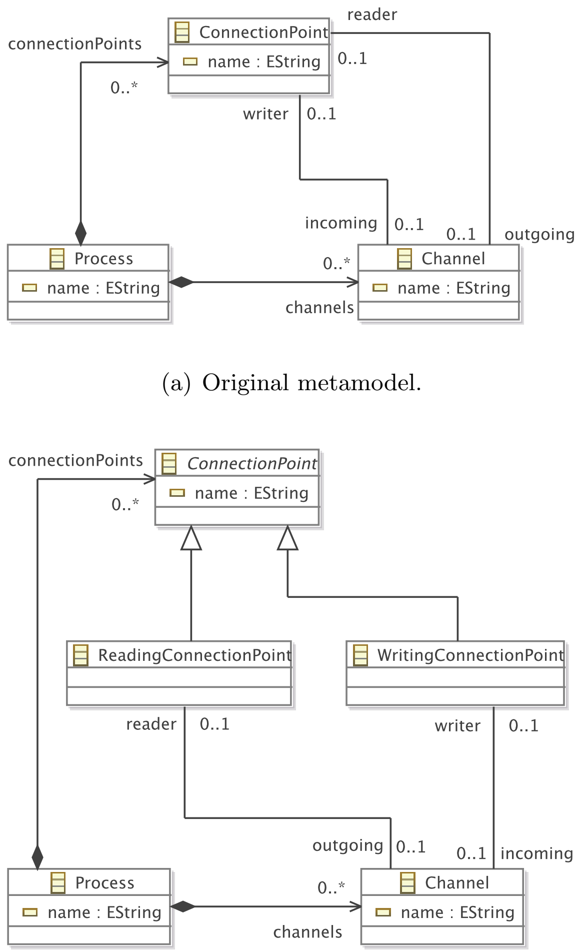
Finally, some categories of model value must be converted before being copied from the original to the migrated model. Again, the need for and semantics of this conversion varies over modelling frameworks. For example, reference values typically require conversion before copying because, once copied, they must refer to elements of the migrated rather than the original model. In this case, the set of equivalences (`<r,e,e'>`) can be used to perform the conversion. In other cases, the target modelling framework must be used to perform the conversion, such as when EMF enumeration literals are copied.

#### Development and User Tools

As discussed in Section 4.2, models and metamodels are typically kept separate. Flock migration strategies can be distributed by the metamodel developer in two ways. An extension point defined by Flock provides a generic user interface for migration strategy execution. Alternatively, metamodel developers can elect to build their own interface, delegating execution responsibility to `FlockModule`. The latter approach facilities interoperability with, for example, model and source code management systems.

### 5.4.2 Examples

Flock is now demonstrated using three examples of model migration. The first demonstrates the way in which a Flock module is executed and illustrates the semantics of conservative copy. The second describes the way in which the migration of the Petri net co-evolution example (introduced above) can be




(a) Original metamodel.



(b) Evolved metamodel.

Figure 5.12: Exemplar Process-Oriented metamodel evolution

specified with Flock, and is included for direct comparison with the other languages discussed in Section 5.3. The final, larger example demonstrates all of the features of Flock, and is based on changes made to UML class diagrams between versions 1.5 and 2.0 of the UML specification.

### Process-Oriented Example

The example presented below demonstrates, in detail, the way in which a Flock module executes a model migration strategy. Consider the original and evolved metamodels shown in Figure 5.12, which are simplifications of two versions of the metamodel from the MDE project described in Appendix B.

The original metamodel, shown in Figure 5.12(a), has been evolved to distinguish between `ConnectionPoints` that are a reader for a `Channel` and `ConnectionPoints` that are a writer for a `Channel` by making `ConnectionPoint` abstract and introducing two subtypes, `ReadingConnectionPoint` and `WritingConnectionPoint`, as shown in Figure 5.12(b).

Suppose that the model shown in Figure 5.13, which conforms to the original metamodel in Figure 5.12(a) is to be migrated. The model comprises thre `Processes` named *delta*, *prefix* and *minus*; three `Channels` named *a*, *b* and *c*; and six `ConnectionPoints` named *a?*, *a!*, *b?*, *b!*, *c?* and *c!*.

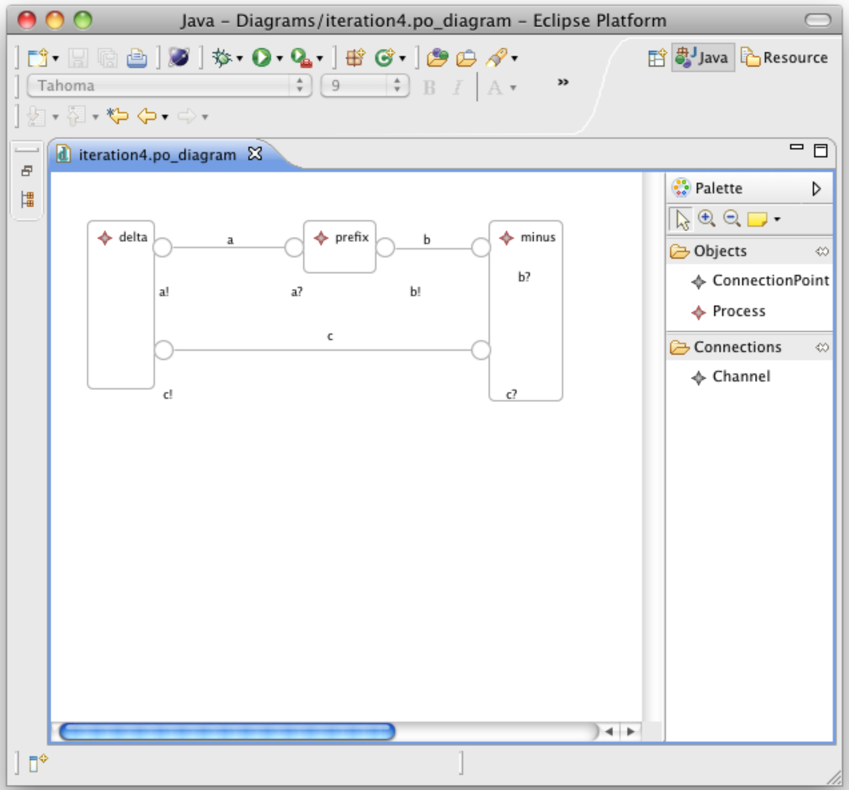



Figure 5.13: Exemplar Process-Oriented model prior to migration

For the migration strategy shown in Listing 5.20, the Flock module will perform the following steps. Firstly, the rule selection phase produces a set of pairs `<r,e>`. For each `ConnectionPoint`, the guard part of the user-defined rules control which rule will be selected. `ConnectionPoints a!`, `b!` and `c!` have outgoing `Channels` (`a`, `b` and `c` respectively) and hence the migration rule on line 1 is selected. Similarly, the `ConnectionPoints a?`,

```
1 migrate ConnectionPoint to ReadingConnectionPoint when: original.
      outgoing.isDefined()
2 migrate ConnectionPoint to WritingConnectionPoint when: original.
      incoming.isDefined()
```

Listing 5.20: Redefining equivalences for the Component model migration.

b? and c? have incoming Channels (a, b and c respectively) and hence the migration rule on line 2 is selected. There is no ConnectionPoint with both an outgoing and an incoming Channel, but if there were, the first applicable rule (i.e. the rule on line 1) would be selected. For the other model elements (the Processes and Channels) no user-defined rules are applicable, and so default rules are used instead. A default rule has an empty body and the identical original and evolved types. In other words, a default rule for the Process class is equivalent to the user-defined rule: migrate Process to Process {}

Secondly, the equivalence establishment phase creates an element, e′, in the migrated model for each pair <r,e>. For each ConnectionPoint, the evolved type of the selected rule controls the type of e′. The rule on line 1 of Listing 5.20 was selected for the ConnectionPoints a!, b! and c! and hence an equivalent element of type ReadingConnectionPoint is created for a!, b! and c!. Similarly, an equivalent element of type WritingConnectionPoint is created for a?, b? and c?. For the other model elements (the Processes and Channels) a default rule was selected, and hence the equivalent model element has the same type as the original model element.

Finally, the rule execution phase performs a conservative copy for each original and equivalent model element in the set of triples <r,e,e′> produced by the equivalent establishment phase. The metamodel evolution shown in Figure 5.12 has not affected the Process type, and hence for each Process in the original model, conservative copy will create a Process in the migrated model and copy the values of all features. For each Channel in the original model, conservative copy will create an equivalent Channel in the migrated model and copy the value of the name feature from original to migrated model element. However, the values of the reader and writer features will not be copied by conservative copy because the type of these features has evolved (from ConnectionPoint to ReadingConnectionPoint and WritingConnectionPoint, respectively). The values of the reader and writer features in the original model might will not conform to the reader and writer features in the evolved metamodel. Finally, the values of the name, incoming and outgoing features of the ConnectionPoint class have not evolved, and hence are copied directly from original to equivalent model elements.

### Petri Nets in Flock

The exemplar Petri net metamodel evolution is now revisited to demonstrate the core functionality of Flock. In Listing 5.21, `Nets` and `Places` are migrated automatically. Unlike the ATL migration strategy (Listing 5.16), no explicit copying rules are required. Compared to the COPE migration strategy (Listing 5.18), the Flock migration strategy does not explicitly unset the original `src` and `dst` features of `Transition`.

```
migrate Transition {
  for (source in original.src) {
   var arc := new Migrated!PTArc;
   arc.src := source.equivalent(); arc.dst := migrated;
   arc.net := original.net.equivalent();
  }

  for (destination in original.dst) {
   var arc := new Migrated!TPArc;
   arc.src := migrated; arc.dst := destination.equivalent();
   arc.net := original.net.equivalent();
  }
}
```

Listing 5.21: Petri nets model migration in Flock

### UML Class Diagrams in Flock

Figure 5.14 illustrates a subset of the changes made between UML 1.5 and UML 2.0. Only class diagrams are considered, and features that did not change are omitted. In Figure 5.14(a), association ends and attributes are specified explicitly and separately. In Figure 5.14(b), the `Property` class is used instead. The Flock migration strategy (Listing 5.22) for Figure 5.14 is now discussed.

```
migrate Association {
  migrated.memberEnds := original.connections.equivalent();
}

migrate Class {
  var fs := original.features.equivalent();
  migrated.operations := fs.select(f|f.isKindOf(Operation));
  migrated.attributes := fs.select(f|f.isKindOf(Property));
  migrated.attributes.addAll(original.associations.equivalent())
}

delete StructuralFeature when: original.targetScope <> #instance

migrate Attribute to Property {
```

```
15   if (original.ownerScope = #classifier) {
16     migrated.isStatic = true;
17   }
18 }
19 migrate Operation {
20   if (original.ownerScope = #classifier) {
21     migrated.isStatic = true;
22   }
23 }
24
25 migrate AssociationEnd to Property {
26   if (original.isNavigable) {
27     original.association.equivalent().navigableEnds.add(migrated)
28   }
29 }
```

Listing 5.22: UML model migration in Flock

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

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

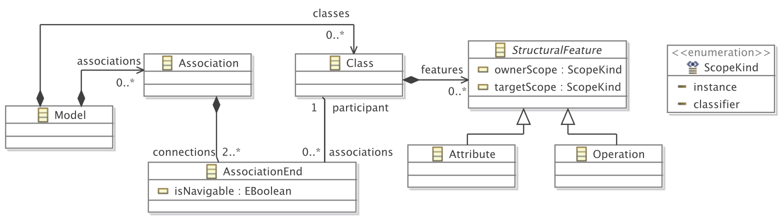
The UML 1.5 `StructuralFeature#targetScope` feature is no longer supported in UML 2.0, and no migration path is provided. Consequently, line 14 deletes any model element whose `targetScope` is not the default value.

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

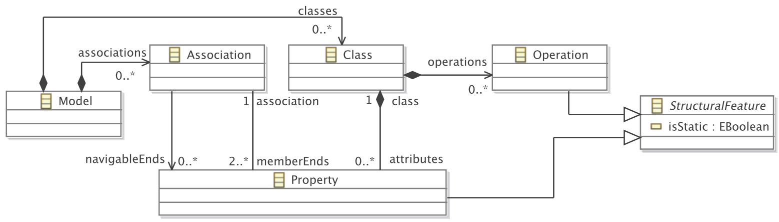
### Summary

Table 5.4.2 illustrates several characterising differences between Flock and the related languages presented in Section 5.3. Due to its conservative copying algorithm, Flock is the only language to provide both automatic copying and unsetting. The evaluation presented in Section 6.2 explores the extent to which automatic copying and unsetting affect the conciseness of migration strategies.

All of the approaches considered in Table 5.4.2 support EMF, arguably the most widely used modelling framework. The Ecore2Ecore approach, however, requires migration to be encoded at the level of the underlying model




(a) Original metamodel.




(b) Evolved metamodel.

Figure 5.14: Exemplar UML metamodel evolution

representation XMI. Both Flock and ATL support other modelling technologies, such as MDR and XML. However, ATL does not automatically copy model elements that have not been affected by metamodel changes. Therefore, migration between models of different technologies with ATL requires extra statements in the migration strategy to ensure that the conformance constraints of the target technology are satisfied. Because it delegates conformance checking to an EMC driver, Flock requires no such checks.

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

| | **Automatic** | | **Modelling** |
|---|---|---|---|
| **Tool** | **Copy** | **Unset** | **technologies** |
| Ecore2Ecore | ✓ | ✗ | XMI |
| ATL | ✗ | ✓ | EMF, MDR, KM3, XML |
| COPE | ✓ | ✗ | EMF |
| Flock | ✓ | ✓ | EMF, MDR, XML, Z |

Table 5.1: Properties of model migration approaches

## 5.5 Chapter Summary

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

To be completed.

# Bibliography


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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[Brooks 1986] Frederick P. Brooks. No silver bullet – essence and accident in software engineering (invited paper). In *Proc. International Federation for Information Processing*, pages 1069–1076, 1986.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[Dig 2007] Daniel Dig. *Automated Upgrading of Component-Based Applications*. PhD thesis, University of Illinois at Urbana-Champaign, USA, 2007.

[Dmitriev 2004] Sergey Dmitriev. Language oriented programming: The next programming paradigm. *JetBrains onBoard [online]*, 1, 2004. [Accessed 30 June 2008] Available at: `http://www.onboard.jetbrains.com/is1/articles/04/10/lop/`.

[Drivalos *et al.* 2008] Nicholas Drivalos, Richard F. Paige, Kiran J. Fernandes, and Dimitrios S. Kolovos. Towards rigorously defined model-to-model traceability. In *Proc. European Conference on the Model Driven Architecture Workshop on Traceability*, 2008.

[Ducasse *et al.* 1999] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Proc. International Conference on Software Maintenance*, pages 109–118. IEEE Computer Society, 1999.

[Eclipse 2008a] Eclipse. Eclipse Modeling Framework project [online]. [Accessed 22 January 2009] Available at: `http://www.eclipse.org/modeling/emf/`, 2008.

[Eclipse 2008b] Eclipse. Eclipse project [online]. [Accessed 20 January 2009] Available at: `http://www.eclipse.org`, 2008.

[Eclipse 2008c] Eclipse. Epsilon home page [online]. [Accessed 30 June 2008] Available at: `http://www.eclipse.org/gmt/epsilon/`, 2008.

[Eclipse 2008d] Eclipse. Generative Modelling Technologies project [online]. [Accessed 30 June 2008] Available at: `http://www.eclipse.org/gmt`, 2008.

[Eclipse 2009a] Eclipse. Model Development Tools project [online]. [Accessed 6 January 2008] Available at: `http://www.eclipse.org/modeling/mdt/`, 2009.

[Eclipse 2009b] Eclipse. UML2 Model Development Tools project [online]. [Accessed 7 September 2009] Available at: `http://www.eclipse.org/modeling/mdt/uml2`, 2009.

[Eclipse 2010] Eclipse. Connected Data Objects Model Repository Project Website [online]. [Accessed 15 February 2010] Available at: `http://www.eclipse.org/modeling/emf/?project=cdo#cdo`, 2010.

[Edelweiss & Freitas Moreira 2005] Nina Edelweiss and Álvaro Freitas Moreira. Temporal and versioning model for schema evolution in object-oriented databases. *Data & Knowledge Engineering*, 53(2):99–128, 2005.

[Elmasri & Navathe 2006] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley Longman, 2006.

[Erlikh 2000] Len Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.

[Evans 2004] E. Evans. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Addison-Wesley, Boston, MA, USA, 2004.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[Kolovos *et al.* 2007a] Dimitrios S. Kolovos, Richard F. Paige, Fiona A.C. Polack, and Louis M. Rose. Update transformations in the small with the Epsilon Wizard Language. *Journal of Object Technology*, 6(9):53–69, 2007.

[Kolovos *et al.* 2007b] Dimitrios S. Kolovos, Richard F. Paige, Louis M. Rose, and Fiona A.C. Polack. Bridging the Epsilon Wizard Language and the Eclipse Graphical Modeling Framework. In *Proc. Eclipse Summit*, Ludwigsburg, Germany, 2007.

[Kolovos *et al.* 2008a] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. The Epsilon Transformation Language. In *Proc. ICMT*, volume 5063 of *LNCS*, pages 46–60. Springer, 2008.

[Kolovos *et al.* 2008b] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. On the evolution of OCL for capturing structural constraints in modelling languages. In *Proc. Workshop on Rigorous Methods for Software Construction and Analysis*, 2008.

[Kolovos *et al.* 2008c] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Scalability : The holy grail of model driven engineering. In *Proc. Workshop on Challenges in Model Driven Engineering*, 2008.

[Kolovos *et al.* 2009] Dimitrios S. Kolovos, Richard F. Paige, and Louis M. Rose. EuGENia: GMF for mortals. Long talk at Eclipse Summit Europe, October 2009, Ludwigsburg, Germany. Available at: `https://www.eclipsecon.org/submissions/ese2009/view_talk.php?id=979` [Accessed 12 April 2010], 2009.

[Kolovos 2009] Dimitrios S. Kolovos. *An Extensible Platform for Specification of Integrated Languages for Model Management*. PhD thesis, University of York, United Kingdom, 2009.

[Kramer 2001] Diane Kramer. XEM: XML Evolution Management. Master's thesis, Worcester Polytechnic Institute, MA, USA, 2001.

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

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

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

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

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

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

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

[Lerner 2000] Barbara Staudt Lerner. A model for compound type changes encountered in schema evolution. *ACM Transactions on Database Systems*, 25(1):83–127, 2000.

[Mäder *et al.* 2008] P. Mäder, O. Gotel, and I. Philippow. Rule-based maintenance of post-requirements traceability relations. In *Proc. IEEE International Requirements Engineering Conference (RE)*, pages 23–32, 2008.

[Martin & Martin 2006] R.C. Martin and M. Martin. *Agile Principles, Patterns, and Practices in C#*. Prentice Hall, Upper Saddle River, NJ, USA, 2006.

[McCarthy 1978] John McCarthy. History of Lisp. *History of Programming Languages*, 1:217–223, 1978.

[McNeile 2003] Ashley McNeile. MDA: The vision with the hole? [Accessed 30 June 2008] Available at: `http://www.metamaxim.com/download/documents/MDAv1.pdf`, 2003.

[Mellor & Balcer 2002] Stephen J. Mellor and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architectures.* Addison-Wesley Longman, 2002.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[Ramil & Lehman 2000] Juan F. Ramil and Meir M. Lehman. Cost estimation and evolvability monitoring for software evolution processes. In *Proc. Workshop on Empirical Studies of Software Maintenance*, 2000.

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

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

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

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

[Rose *et al.* 2009a] Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Enhanced automation for managing model and metamodel inconsistency. In *Proc. ASE*, pages 545–549. ACM Press, 2009.

[Rose *et al.* 2009b] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona A.C. Polack. An analysis of approaches to model migration. In *Proc. Joint MoDSE-MCCM Workshop*, 2009.

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

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

[Rose *et al.* 2010c] Louis M. Rose, Dimitrios S. Kolovos, Nicholas Drivalos, James. R. Williams, Richard F. Paige, Fiona A.C. Polack, and Kiran J. Fernandes. Concordance: An efficient framework for managing model integrity [submitted to]. In *Proc. European Conference on Modelling Foundations and Applications*, volume 6138 of *LNCS*, pages 62–73. Springer, 2010.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[Welch & Barnes 2005] Peter H. Welch and Fred R. M. Barnes. Communicating mobile processes. In *Proc. Symposium on the Occasion of 25 Years of Communicating Sequential Processes (CSP)*, volume 3525 of *LNCS*, pages 175–210. Springer, 2005.

[Winkler & Pilgrim 2009] Stefan Winkler and Jens von Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Software and Systems Modeling*, December 2009.