Chapter 4

Analysis

The literature review presented in Chapter 3 motivated a deeper analysis of existing techniques for managing and identifying the effects of evolution in the context of MDE. Figure 4.1 summarises the objectives of this chapter. On the left are the artefacts used to produce those on the right. As shown in Figure 4.1, 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).

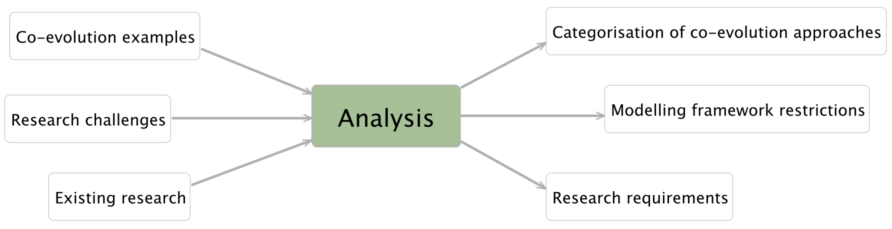


Figure 4.1: Analysis chapter overview.

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

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).

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

NameFoo	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

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. I considered two zoos (the Atlantic Zoo and the AtlantEcore Zoo[2]), but neither contained any significant external metamodel changes. Those changes that were made involved only renaming of meta-classes (trivial to migrate) or additive changes (which do not affect consistency, 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 I developed an implementation of FPTC in Eclipse. The implementation includes an FPTC metamodel. Recent work with Philippa Conmy, a research assistant in the department, has identified a significant flaw in the implementation, leading to changes to the metamodel. These changes caused existing FPTC models to become inconsistent with the metamodel. Conmy has made

[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/`

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 cannot 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 caused inconsistency with existing models. xText provides examples of use, 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 caused inconsistency with 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). GMF does, however, have a large number of users, and it might be possible to gather this information from those users. However, contacting GMF users and analysing any data they might supply was not possible in this thesis, due to time constraints. **Suitability for study:** Suitable for studying co-evolution. Unsuitable for studying synchronisation, unless extra data is obtained from GMF users.

Summary of selection

The FPTC and xText projects were selected for a study of co-evolution. No appropriate projects were located for a study of synchronisation. The GMF

[4]Private communication, 2008.

project will not be studied immediately, but reserved for evaluation (Chapter 6).

4.1.3 Other examples

Because only a small number of MDE projects fulfilled all of the requirements, additional data was collected from alternative sources. Firstly, examples were sought from object-oriented systems, which have some similarities to systems developed using MDE. Secondly, examples were discovered during 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 non-breaking in EMF, which automatically provides values for the inverse of a reference when required.

3. **Domain-specific refactorings** manage issues specific to object-oriented programming, such as casting, defensive return values, and assertions. These issues are not relevant to metamodelling.

The object-oriented refactorings that can be applied to metamodels provide examples of metamodel evolution. When applied, some of these refactorings potentially cause inconsistency between a metamodel and its models. By using Fowler's description of each refactoring, a migration strategy for updating (co-evolving) inconsistent models was deduced. An example of this process is now presented.

Figure 4.2 illustrates a refactoring that changes a reference object to a value object. 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.

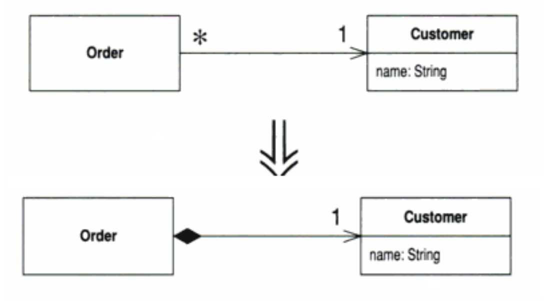


Figure 4.2: Refactoring a reference to a value. Taken from [Fowler 1999][pg183].

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 metamodelling, and caused inconsistencies between a metamodel and its models.

```
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 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 prototypical metamodel to standardise the way in which process-oriented programs are modelled was produced with Adam Sampson, a research assistant at the University of Kent, and an investigation of the feasibility of implementing a tool for generating story-worlds for interactive narratives was conducted with Heather Barber, a postdoctoral researcher in the 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

To summarise, in this section the example data used to analyse existing structures and processes for managing evolution in the context of MDE was discussed. Example data was sought from existing MDE projects, and also from a related domain and research collaboration. Eight existing 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 as a case study and is used for evaluation 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, this thesis now focuses on model and metamodel co-evolution.

4.2 Analysing Existing Techniques

The examples of co-evolution identified in the previous section were analysed to discover and compare existing techniques for managing co-evolution. Details of this process are included in Appendix A. 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) . These results have been published in [Rose *et al.* 2009b, Rose *et al.* 2010f].

4.2.1 Modelling Framework Characteristics Relevant to Co-Evolution

Analysis of the co-evolution examples identified in the previous section highlights characteristics of modern MDE modelling development environments that impact on 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, MDE 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 conforms 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. Executable migration strategies can only be used if they are specified with a tool that does not depend on the modelling framework to load the non-conformant models (and, at present, no such tool exists). Typically then, the metamodel user can only perform migration by editing the model directly, normally manipulating its underlying representation (e.g. XMI).

Because modelling frameworks do not permit changes to a model that cause non-conformance, model migration must produce a model that conforms to the evolved metamodel. Therefore, model migration cannot be specified as a combination of co-evolution techniques with each performing some part of the migration, because intermediate steps are not allowed to leave the model in a non-conformant state.

Finally, a further consequence of implicitly enforced conformance is that models cannot be checked for conformance against any metamodel other than their own. Because conformance is always assumed, modern MDE development environments provide limited mechanisms for checking conformance, and typically provide no support for checking conformance to a metamodel other than the one used to construct the model.

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.* No existing research has explored user-driven co-evolution, yet analysis of the co-evolution examples identified in Section 4.1 highlighted several instances of user-driven co-evolution. This section discusses user-driven co-evolution and provides a scenario (based on examples from Section 4.1).

In user-driven co-evolution, the metamodel user performs migration by loading their models to test conformance, and then rectifying 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. This is analogous to developing a legacy system[5].

Users of the same metamodel migrate their models independently. When no migration strategy has been provided by the metamodel developer, each metamodel user must devise their own migration strategy.

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, in particular, is regarded as sub-optimal for use by humans [OMG 2004]) and, consequently, using them for migration is an unproductive and tedious task. When directly editing the underlying format of a model, re-establishing consistency is often a slow, iterative process. For example, EMF uses a multi-pass model parser and hence only reports one category of errors when a model cannot be loaded.

[5]A system for which one has no documentation and whose authors have left the owning organisation. TODO - Fiona suggested reviewing legacy systems in the literature review

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 migration. Firstly, the underlying model representation is unlikely to be optimised for human usability. Together with limited support for conformance checking, user-driven migration performed by editing the underlying model representation is error prone and tedious. Secondly, installing a new version of a metamodel plug-in can affect the conformance of models and, moreover, 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. Metamodels such as those defined in UML, EMF, MOF and GMF have many more users and, hence, user-driven co-evolution is arguably less desirable than developer-driven co-evolution.

4.2.3 Categorisation of Developer-Driven Co-Evolution Techniques

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. Three categories of developer-driven co-evolution approach were identified: *manual specification*, *operator-based* and *metamodel matching*. This categorisation has been published in [Rose *et al.* 2009b] and revisited in [Rose *et al.* 2010f]. 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 metamodel matching 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 these patterns in the chosen migration 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 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. Wachsmuth [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.

The usefulness of an operator-based co-evolution approach depends heavily on the richness of the library of co-evolutionary operators that it provides. If the library of co-evolutionary operators cannot be extended, the metamodel developer must use another approach for performing model migration when no appropriate co-evolutionary operator is available. COPE allows metamodel developers to manually specify custom migration strategies when no co-evolutionary operator is appropriate, using a general purpose programming language. (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). A custom migration strategy can be imported as a co-evolutionary operator if it can be specified independently of its metamodel. Importing an operator in COPE requires the metamodel developer to rewrite their custom migration strategy, removing references to their metamodel.

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 that a metamodel developer may wish to perform. 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[6], is shown below.

> **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.

COPE operators are defined in Groovy[7]. To select the correct operator, users can read descriptions (an example is shown above), examine the source code, or try executing the operator (an undo command is provided). There

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

[7]A dynamic, strongly typed object-oriented programming language for the Java platform

are no formally defined semantics for COPE's operators, and verifying their effects involves inspecting the model on which they are executed.

Other techniques can be used to try to improve the navigability of large libraries of co-evolutionary operators. COPE, for example, restricts the choice of operators to only those that can be applied to the currently selected metamodel element. 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 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.

Metamodel Matching

In *metamodel matching*, a migration strategy is inferred by analysing the evolved metamodel and the *metamodel history*. Metamodel matching approaches use one of two categories of metamodel history; either the original metamodel (*differencing* approaches) or the changes made to the original metamodel to produce the evolved metamodel (*change recording* approaches). 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[8] to produce a model-to-model transformation from the difference model. Cicchetti et al. [Cicchetti *et al.* 2008] and Garcés et al. [Garcés *et al.* 2009] describe metamodel matching approaches. More specifically, both describe differencing 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.

Compared to manual specification and operator-based co-evolution, metamodel matching requires 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,

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

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

Because metamodel matching 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 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.

Before discussing the benefits and limitations of differencing and change recording metamodel matching approaches, an example of co-evolution is introduced.

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[9]. Figure 4.3 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.

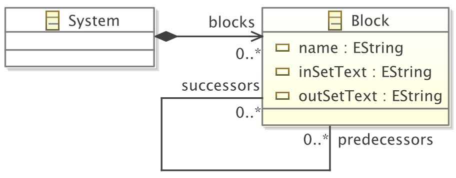


Figure 4.3: Original metamodel, prior to evolution [Rose *et al.* 2009b].

Further analysis of the domain revealed that extra information about the relationship between `Blocks` needed to be stored. The evolved metamodel is shown in Figure 4.4. The `Connection` class is introduced to capture this extra information. `Blocks` are no longer related directly to `Blocks`, instead they are related via an instance of the `Connection` class. The

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

incomingConnections and outgoingConnections references of Block are used to relate Blocks to each other via an instance of Connection.

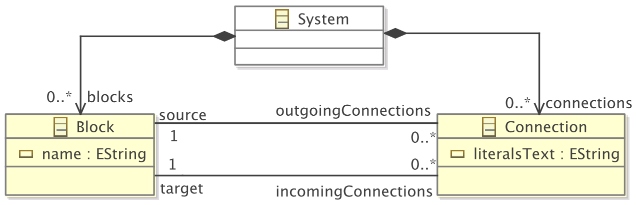


Figure 4.4: Evolved metamodel with Connection metaclass [Rose *et al.* 2009b].

A model that conforms to the original metamodel (Figure 4.3) might not conform to the evolved metamodel (Figure 4.4). Below is a description of 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.

Differencing and change recording metamodel matching approaches are now compared and contrasted.

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, or Change the type of feature successors from Block to Connection). The record of changes may be reduced to a normal form, which might remove redundancy, but might also erase useful information. In change recording, some types of metamodel evolution can be more easily recognised than with differencing. With change recording, renaming of a metamodel element from X to Y can be distinguished from the following sequence: remove a metamodel element called X, add a metamodel element called Y. 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 4.4) could have been produced by changing the name and type of the `successors` reference (shown in Figure 4.3). 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`. 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. In his thesis, [Cicchetti 2008] discusses this issue, devising a normal form, optimised for his approach, 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) and evolved (Figure 4.4) 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 discovery of user-driven, rather than developer-driven, model migration and to a categorisation of 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 migration). Because modelling frameworks implicitly enforce conformance, the underlying representation of non-conformant models is edited by the metamodel user when no executable migration strategy is provided by the metamodel developer (user-driven migration).

User-driven migration, which has not yet been explored in existing research, 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 migration is required.

In Section 4.1, existing techniques for performing developer-driven co-evolution were compared, contrasted and categorised. The categorisation highlights a trade-off between flexibility and effort for the metamodel-developer when choosing between categories of approach. 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, metamodel matching approaches seek to infer a migration strategy from a metamodel history and hence require less effort from the metamodel developer. However, a metamodel matching approach affords the metamodel developer less flexibility, and may restrict the metamodel evolution process. (For example, the order or way in which the metamodel is changed may influence the inference of the migration strategy). Operator-based approaches occupy the middle-ground: by restricting the way in which metamodel evolution is expressed, an operator-based approach can be used to infer a migration strategy. The metamodel developer selects appropriate operators that express both metamodel evolution and model migration. Operator-based approaches require a specialised metamodel editor, and it is not yet clear whether they can be applied when a metamodel is represented with a freeform (e.g. textual) rather than a structured (e.g. tree-based) syntax.

4.3 Requirements Identification

In Chapter 3, research objectives were identified. Based on the analysis presented in this chapter, the objectives were refined, deriving requirements for this thesis. [10]

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

4.3.1 Explicit conformance checking

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

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

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

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 user-driven co-evolution techniques, highlighting several of the challenges faced when they are applied. Those challenges are now summarised and used to derive requirements for this thesis.

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

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

4.3.3 Developer-driven co-evolution

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

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

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

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

A domain-specific language (discussed in Chapter 3) provides one way to capture re-occurring patterns. When accompanied with an execution engine that encapsulates infrastructure features, a domain-specific language is a common way for specifying model management operations in modern model-driven development environments. Domain-specific languages are provided by model-to-model (M2M) transformation tools such as ATL [ATLAS 2007], VIATRA [**?**], workflow architectures such as oAW [openArchitectureWare 2007], and model-to-text (M2T) transformation tools such as MOFScript [Oldevik *et al.* 2005] and XPand [openArchitectureWare 2007].

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

4.4 Chapter Summary

To be completed.

Chapter 5

Implementation

Section 4.3 identified requirements for structures and processes for managing co-evolution. In this chapter, the way in which this thesis approaches those requirements is described. Several related solutions were implemented, using domain-specific languages, automation and extensions to existing modelling technologies. Figure 5.1 summarises the structure of the chapter. To better support co-evolution and to overcome restrictions with existing modelling frameworks, a metamodel-independent syntax was devised and implemented, enabling model and metamodel decoupling and consistency checking (Section 5.1). To address some of the challenges faced in user-driven co-evolution, an OMG specification for an alternative, textual modelling notation was implemented (Section 5.2). Model migration languages were identified, analysed and compared, leading to the derivation and implementation of a new model transformation language tailored for model migration and centred around a novel approach to relating source and target model elements (Section 5.4).

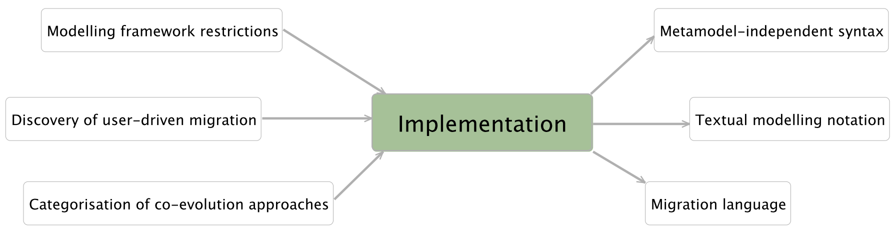

Modelling framework restrictions
Discovery of user-driven migration
Categorisation of co-evolution approaches
Implementation
Metamodel-independent syntax
Textual modelling notation
Migration language

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. Because of this, modelling frameworks cannot be used to load non-conformant models, and provide little support for checking the conformance of a model with other metamodels or other versions of a metamodel. 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 storage representation 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 storage representation 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 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" />
</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`.

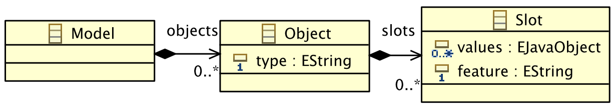


Figure 5.2: A generic metamodel.

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.

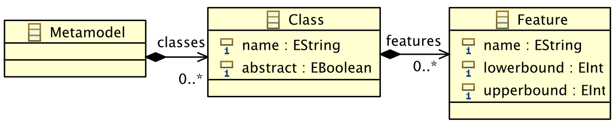


Figure 5.3: Minimal MOF metamodel.

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. EMF, for example, is implemented in Java and exposes some services for checking the type compatibility of model data with metamodel features. All metamodel features are typed and their types provide methods for determining the underlying programming language representation. Type compatibility checks can be implemented using these methods.

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 Objects, all with type "Person". Each class object contains a slot whose feature is name, one with the value "Franz", one with the value "Julie" and the other with 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[1] to the object that contains the name slot with the value "Julie" and one whose feature is father and whose value is a reference to the object containing "Hermann". A UML object diagram for this instantiation of the generic metamodel is shown in Figure 5.4. Instances of object (slot) are shaded grey (white).

[1]The generic metamodel used in this thesis implements reference values using the proxy design pattern [Gamma *et al.* 1995].

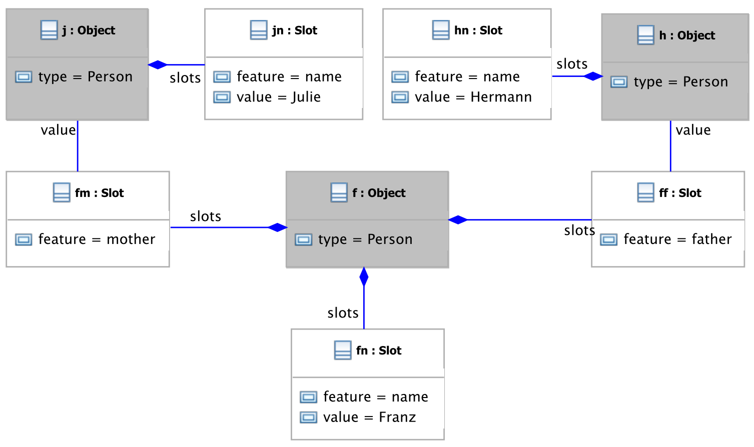


Figure 5.4: Exemplar instantiation of generic metamodel.

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 that: 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 model representation discussed here. In Section 5.4, a domain-specific language for migration 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.* 2009b], proposes a model migration language that manipulates models via a metamodel-independent syntax. Some of the strengths and weaknesses of that approach and 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 particularly useful during metamodel installation. As discussed in Section 4.2.1, metamodel developers do not have access to downstream models. Consequently, instances of a metamodel may become non-conformant after a new version of a metamodel plug-in is installed. By default, an EMF metamodel plug-in does not check conformance during plug-in installation and non-conformant models are only detected when the user attempts to load them.

To enable conformance checking as part of metamodel installation in EMF, the binding to a generic metamodel discussed above has been integrated with Concordance [Rose *et al.* 2010c] in joint work with Dimitrios S. Kolovos, a lecturer in this department, Nicholas Drivalos, a research associate in this department and James R. Williams, a research student in this department.

Concordance provides a light-weight and efficient mechanism for resolving inter-model references, including the references between models and their metamodels. Concordance can be used to efficiently determine the instances of a metamodel, which is otherwise only possible with a brute force search of a development workspace.

The integration work involved extending Concordance such that, after the installation of a metamodel plug-in, models that conform to any previous version of the metamodel are identified. Those models are checked for conformance with the new metamodel. As such, conformance checking occurs automatically and during metamodel installation. Conformance problems are detected and reported immediately, rather than when the user next attempts to load an affected model. By integrating conformance checking with Concordance, improved scalability is achieved, as demonstrated in [Rose *et al.* 2010c].

5.2 Textual Modelling Notation

The analysis of co-evolution examples in Chapter 4 highlighted two categories of process for managing co-evolution, developer-driven and user-driven. In the former, migration strategies are executable, while in the latter they are not. 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 work on implementing the HUTN specification by Muller and Hassenforder [Muller & Hassenforder 2005] has been abandoned in favour of Sintaks [IRISA 2007], which operates on domain-specific concrete syntax.

Model storage representations are often optimised to reduce storage space or to increase 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. Because no reference implementation exists, the specification does not evaluate the human-usability of the notation. This thesis proposes that HUTN be used instead of XMI for user-driven co-evolution. Further discussion of the human-usability of HUTN is deferred to Chapter 6.

Like the generic metamodel presented in Section 5.1, HUTN is a metamodel-independent syntax for MOF. In this section, the core syntax and key features of HUTN are introduced. The complete definition is available in [OMG 2004]. To illustrate usage of the notation, the MOF-based metamodel of families in Figure 5.5 is used. (A nuclear family "consists only of a father, a mother, and children." [Merriam-Webster 2010]).

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

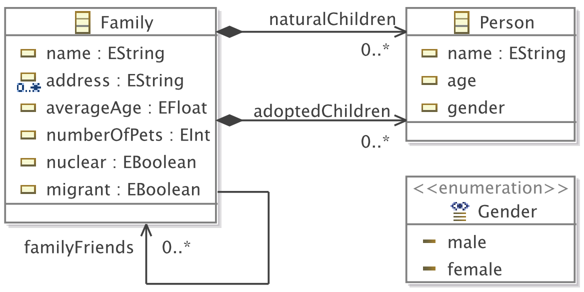


Figure 5.5: Exemplar families metamodel

classes to be constructed (`FamilyPackage`) and a corresponding identifier (`families`), used for fully-qualifying 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, or this HUTN document is configured to require unique identifiers over the entire document.

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 (not yet implemented); renaming of metamodel elements; giving default values for attributes; and stating an attribute whose values are used to infer a default identifier.

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 [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,model-to-text transformation and model validation 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.

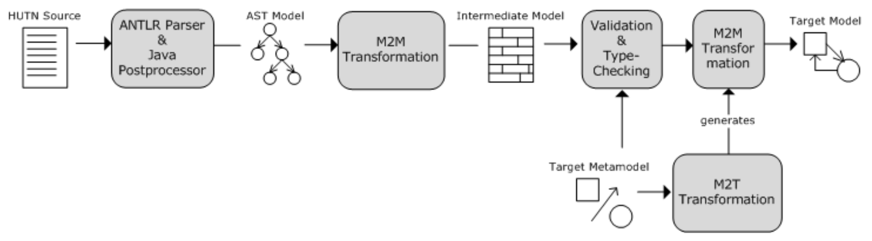


Figure 5.6: The architecture of Epsilon HUTN.

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 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, from the generic metamodel to 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 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
 3     to p : Intermediate!PackageObject {
 4
 5     guard : n.parent.isUndefined()
 6
 7     p.type := n.text;
 8     p.line := n.line;
 9     p.col := n.column;
10
11     var slot := new Intermediate!ContainmentSlot;
12     for (child in n.children) {
13        slot.objects.add(child.equivalent());
14     }
15     if (slot.objects.notEmpty()) {
16        p.slots.add(slot);
17     }
18  }
```

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.allInstances()
                .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 executed on the model at this stage. For this purpose, the conformance constraints are specified in EVL.

Intermediate Model to Target Model Because the contextual analysis is performed on the intermediate model, models conform to 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 an EGL template on the target metamodel. EGL is a template-based text generation 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 is the EGL template for a M2T transformation on the target metamodel; it generates the M2M transformation used for generating the target model. The loop beginning on line 1 iterates over each meta-class in the metamodel, producing a transformation rule to generate target model instances of that meta-class from class objects in the intermediate model. The template guard (line 6) specifies that only class objects of the same type as the meta-class be transformed by the current rule. For the body of the rule the template 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 template body 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] {

  guard: o.type = '[=class.name]'

  -- body omitted
 }
[
```

Listing 5.11: Initial sections of the template (in EGL) for generating rules (in ETL) to instantiate classes of the target metamodel.

Presently, Epsilon HUTN can be used only to generate EMF models. Support for other modelling languages, such as MDR, 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

Epsilon HUTN uses the generic metamodel (from Section 5.1) as an intermediary, facilitating transformation from XMI to HUTN (i.e. the inverse of the transformation discussed above): XMI is parsed to produce an instance of the generic metamodel, and an unparser (implemented using the visitor design pattern [Gamma *et al.* 1995]) generates HUTN source. In this manner, HUTN can be generated for any XMI document, regardless of whether the model described by the XMI conforms to its metamodel.[2]

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 5.12. Recall that the XMI describes three Persons, Franz, Julie and Hermann. Julie and Hermann are the mother and father of Franz.

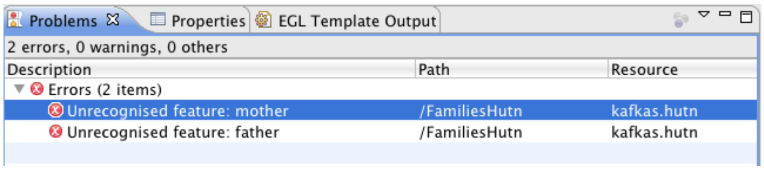
```
Persons "kafkas" {
   Person "Franz" { name: "Franz" }
   Person "Julie" { name: "Julie" }
   Person "Hermann" { name: "Hermann" }

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

Listing 5.12: 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.

[2]TODO: Somewhere, I need to discuss loss of information. (e.g. model element type information when a metaclass is removed)

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.



2 errors, 0 warnings, 0 others

Description	Path	Resource
Errors (2 items)		
Unrecognised feature: mother	/FamiliesHutn	kafkas.hutn
Unrecognised feature: father	/FamiliesHutn	kafkas.hutn

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 5.13 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 5.13: HUTN for people with parents.

5.2.4 Limitations

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 5.14 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 5.14: 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, there might still be benefits from using HUTN in such cases, if the metamodel is likely to be modified frequently, of it the model does not yet have a formal metamodel.

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

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, and an implementation is described in the sequel. The work described in this section has been published in [Rose *et al.* 2010f].

5.3.1 Co-Evolution Example

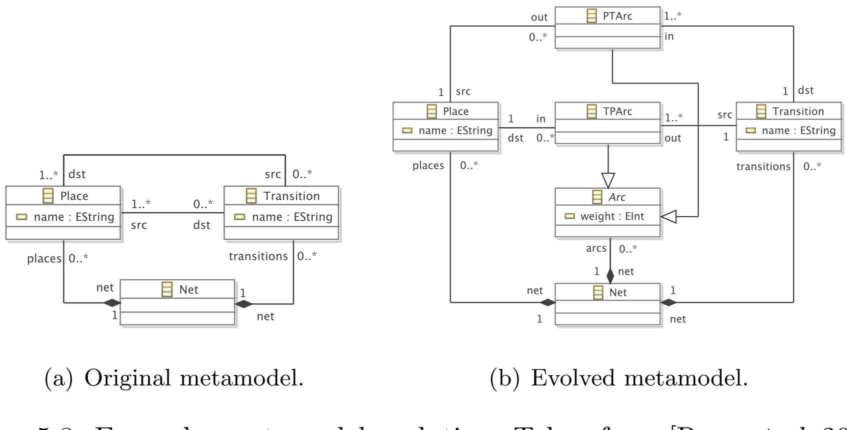
Throughout this section, the following example of an evolution of a Petri net metamodel is used to discuss co-evolution and model migration. The same example has been used previously in co-evolution literature [Cicchetti *et al.* 2008, Garcés *et al.* 2009, Wachsmuth 2007].

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:

```
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.
```



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

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

```
        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.

Using the above example, the existing approaches for specifying and executing model migration strategies are now compared.

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 5.15. Rules for migrating `Places` and `TPArcs` have been omitted for brevity, but are similar to the `Nets` and `PTArcs` rules.

In ATL, *rule*s 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 5.15 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).

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.

```
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 5.15: Fragment of the Petri nets model migration in ATL

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 5.15) and the `Places` rule (not shown) exist only for this reason.

The `Transitions` rule in Listing 5.15 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`.

Manual Specification with Ecore2Ecore Mapping

Hussey and Paternostro [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.

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 5.16 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 5.16: 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 editor. Operators may be applied to an EMF metamodel, and a record of changes tracks their application. Once metamodel evolution is complete, a migration strategy can be generated automatically from the record of changes. 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 metamodels using a *metamodel-independent representation*. Figure 5.9 shows a simplification of the metamodel-independent representation used by COPE. By using a metamodel-independent representation of models as an intermediary, an existing target transformation can be used for performing model migration when the migration strategy is specified in terms of the metamodel-independent representation. Further details of this technique are given in [Herrmannsdoerfer *et al.* 2009b].

Listing 5.17 shows the COPE model migration strategy for the Petri net example given above[3]. Most notably, slots for features that no longer exist

[3]In Listing 5.17, some of the concrete syntax has been changed in the interest of read-

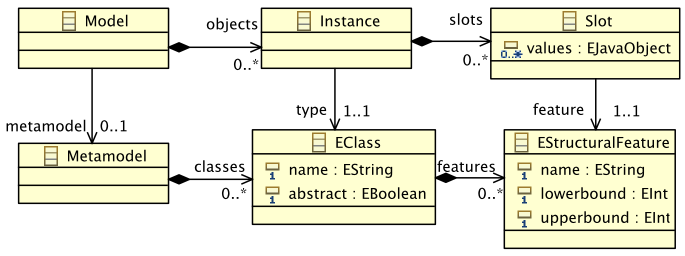


Figure 5.9: Simplification of the metamodel-independent representation used by COPE, based on [Herrmannsdoerfer *et al.* 2009b].

must be explicitly `unset`. In Listing 5.17, slots are `unset` on four occasions, 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.

```
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 5.17: Petri nets model migration in COPE

5.3.3 Requirements Identification

By analysing 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 dimensions: the source-target relationship of the

ability.

language used for specifying migration strategies and the way in which models are represented during migration.

Source-Target Relationship

New target transformation languages (Section 5.3.2) require code for explicitly copying from the original to the evolved metamodel those model elements that are unaffected by the metamodel evolution. In contrast, model migration strategies written in COPE (Section 5.3.2) must explicitly unset any data that is not to be copied from the original to the migrated model. The Ecore2Ecore approach (Section 5.3.2) 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 configured by hand or, in some cases, automatically derived.

In each case, extra effort is required when defining a migration strategy due to the way in which the co-evolution approach relates source (original) and target (migrated) model elements. This observation led to the following requirement: *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 via XMI. Consequently, the metamodel developer must be familiar with XMI and must perform tasks such as dereferencing URI fragments (Listing 5.16) and type conversion. With COPE and the Epsilon Transformation Language, models are loaded using a modelling framework (and so migration strategies need not be concerned with the representation used to store models). 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. Like, the Ecore2Ecore approach, COPE can be used only to manage co-evolution for models and metamodels specified with EMF. Tight coupling to EMF allows the Ecore2Ecore approach to schedule migration automatically, during model loading. 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, including the way in which automatically maps each element of the original model to an equivalent element of the migrated model using a novel conservative copying algorithm and user-defined migration rules. In Section 5.4.2, Flock is demonstrated via application to two examples of model migration. The work described in this section has been published in [Rose *et al.* 2010f].

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 (5.2.2), Flock is built atop Epsilon, which was described in Section 2.3.2. In particular, Flock uses the model connectivity layer and the object language of Epsilon. The former is used to decouple migration from the representation of models and to provide compatibility with several modelling frameworks, while the latter provides a language for specifying user-defined migration rules, which are now discussed.

Abstract Syntax

As illustrated by Figure 5.10, Flock migration strategies are organised into modules (`FlockModule`), which inherit from EOL modules (`EolModule`), which provides support for module reuse with import statements and user-defined operations. 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 5.18 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`

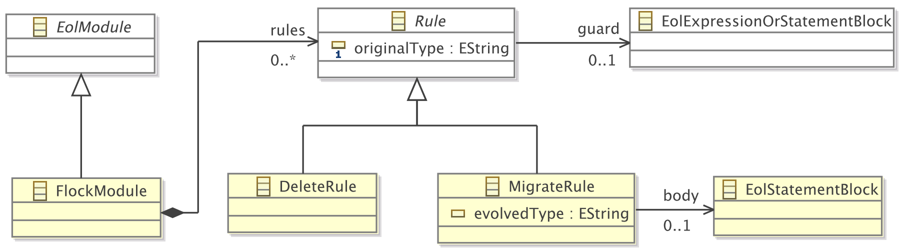


Figure 5.10: The abstract syntax of Flock.

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

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

Listing 5.18: 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 there is presently no create rule. In Flock, the creation of new model elements is usually encoded in the imperative part of a migrate rule specified on the containing type.

Execution Semantics

A Flock module has the following behaviour when executed:

1. For each original model element, `e`:

   Identify an applicable rule, `r`. To be applicable for `e`, a rule must have as its original type the metaclass (or a supertype of the metaclass) of `e` and the guard part of the rule must be satisfied by `e`.

   When no rule can be applied, a default rule is used, which has the metaclass of `e` as its original type, and an empty body.

2. For each mapping between original model element, `e`, and applicable delete rule, `r`:

   Do nothing.

3. For each mapping between original model element, `e`, and applicable migrate rule, `r`:

Create an equivalent model element, `e′` in the migrated model. The metaclass of `e′` is determined from 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).

4. For each mapping between original model element, `e`, applicable migrate rule, `r`, and equivalent model element, `e′`:

   Execute the body of `r` binding `e` and `e′` to variables named `original` and `migrated`, respectively.

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. Because of its conservative copy algorithm, Flock is a hybrid of new target and existing target transformation languages. This section discusses the conservative copying algorithm in more detail.

The algorithm operates on an original model element, `o`, and its equivalent model element in the migrated model, `e`. When `o` has no equivalent in the migrated model (for example, when a metaclass has been removed and the migration strategy specifies no alternative metaclass), `o` is not copied to the migrated model. Otherwise, conservative copy is invoked for `o` and `e`, proceeding as follows:

- For each metafeature, `f` for which `o` has specified a value

  Locate a metafeature in the evolved metamodel with the same name as `f` for which `e` may specify a value.

  When no equivalent metafeature can be found, do nothing.

  Otherwise, copy to the migrated model the original value (`o.f`) only when it conforms to the equivalent metafeature

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 during the development of Flock. The connectivity layer now provides a conformance checking service. Each EMC driver was extended to include conformance checking semantics specific to its modelling framework. Flock implements conservative copy by delegate 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. Reference values typically require conversion before copying. In this case, the mappings between original and migrated model elements maintained by the Flock runtime 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 to be copied.

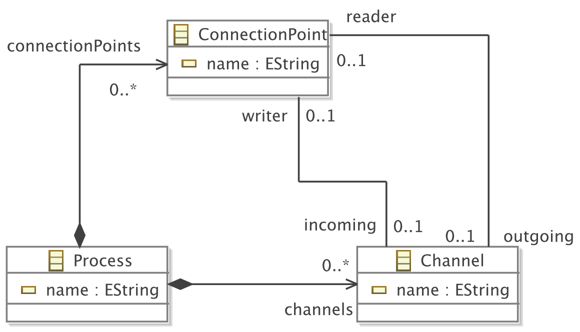
Conservative Copy Example An example is now used to illustrate the semantics of conservative copy. Consider the original and evolved metamodels shown in Figure 5.11, which are simplifications of two versions of the metamodel from the MDE project described in Appendix B.

The original metamodel, shown in Figure 5.11(a), has been evolved to distinguish between `ConnectionPoints` that are a source for a `Channel` and `ConnectionPoints` that are a target for a `Channel` by making `ConnectionPoint` abstract and introducing two subtypes, `ReadingConnectionPoint` and `WritingConnectionPoint`, as shown in Figure 5.11(b).

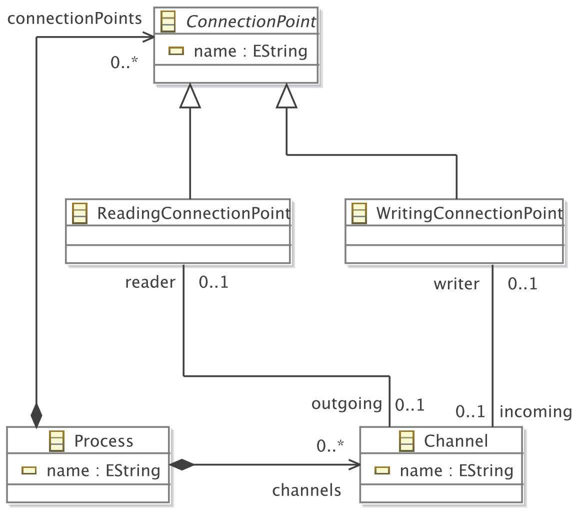
Suppose that the model shown in Figure 5.12, which conforms to the original metamodel (Figure 5.11(a)) is to be migrated. The model comprises thre `Processes` named *delta*, *prefix* and *minus*; three `Channels` named *a*, *b* and *c*; and six `ConnectionPoints` named *a?*, *a!*, *b?*, *b!*, *c?* and *c!*.

Conservative copy takes as input the original model (Figure 5.12) and a set of equivalences over original and evolved metamodel types. The set of equivalences is derived from user-defined `migrate` rules in the Flock source code, as described in Section 5.4.1. For now, let us assume that no user-defined `migrate` rules have been provided and, consequently, equivalences are established using type names (i.e. the equivalent of `Process` is `Process`, the equivalent of `Channel` is `Channel`, etc).

Conservative copy is executed once for each element of the original model, creating an equivalent element in the migrated model and copying values from original to migrated model. The metamodel evolution shown in Figure 5.11 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.



(a) Original metamodel.



(b) Evolved metamodel.

Figure 5.11: Exemplar Process-Oriented metamodel evolution

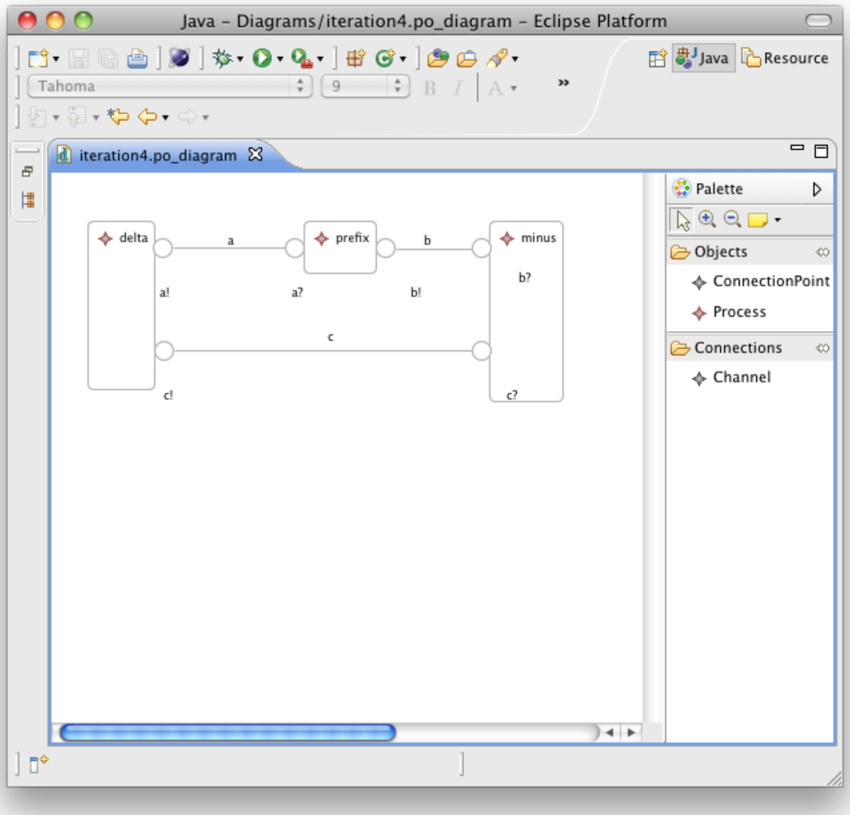


Figure 5.12: Exemplar Process-Oriented model prior to migration

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 during metamodel evolution the type of these features has been changed (from `ConnectionPoint` to `ReadingConnectionPoint` and `WritingConn-ectionPoint`, respectively).

The `ConnectionPoint` type has become abstract during metamodel evolution and can no longer be instantiated. Therefore, for each `Connectio-nPoint` in the original model, conservative copy will not create an equivalent `ConnectionPoint` in the migrated model.

As described in Section 5.4.1, user-defined rules can control the set of equivalences used by conservative copy. Suppose the rules shown in Listing 5.19 are used to migrate the component model shown in Figure 5.12.

From these rules, Flock will suggest `ReadingConnectionPoint` or `Wr-`

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

Listing 5.19: Redefining equivalences for the Component model migration.

itingConnectionPoint as an equivalent type for ConnectionPoints in the original model. Consequently, conservative copy will create an ReadingConnectionPoint or WritingConnectionPoint for each ConnectionPoint in the original model and copy the values of the name, incoming and outgoing features, which have not been affected by metamodel evolution.

Ask Richard and Fiona: Would this section be clearer if I represented the set of equivalences as a relation? I think of it as set of triples: source, target and a guard (which defaults to true). Perhaps it might be easier to explain as a relation with a set of conditions?

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. We anticipate the latter to be useful for production environments using model or source code management repositories.

5.4.2 Examples

Flock is now demonstrated using two examples of model migration. Listing 5.20 illustrates the Flock migration strategy for the Petri net example introduced above and is included for direct comparison with other approaches. An additional, larger example is presented based on changes made to UML class diagrams between versions 1.5 and 2.0 of the UML specification.

Petri Nets in Flock

The exemplar Petri net metamodel evolution is now revisited to demonstrate the basic functionality of Flock. In Listing 5.20, Nets and Places are migrated automatically. Unlike the ATL migration strategy (Listing 5.15), no explicit copying rules are required. Compared to the COPE migration strategy (Listing 5.17), the Flock migration strategy does not explicitly unset the original src and dst features of Transition.

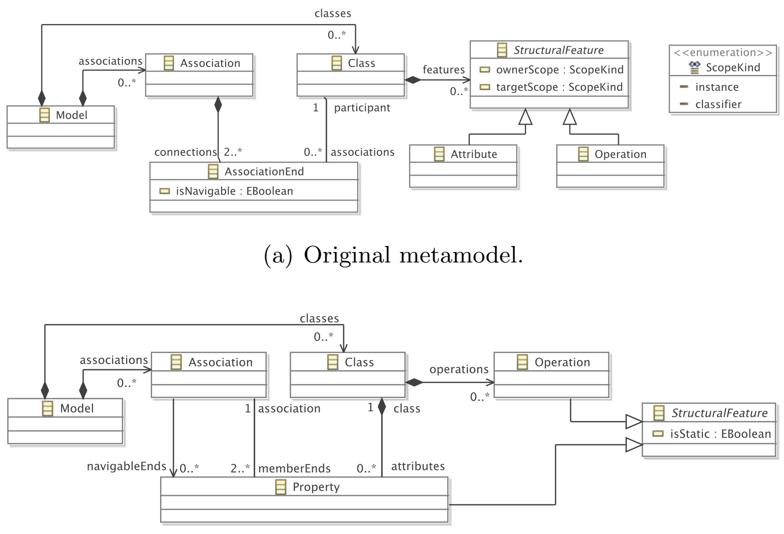
```
migrate Transition {
  for (source in original.src) {
    var arc := new Migrated!PTArc;
    arc.src := source.equivalent(); arc.dst := migrated;
    arc.net := original.net.equivalent();
  }

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

Listing 5.20: Petri nets model migration in Flock

UML Class Diagrams in Flock

Figure 5.13 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 5.13(a), association ends and attributes are specified explicitly and separately. In Figure 5.13(b), the `Property` class is used instead. The Flock migration strategy (Listing 5.21) for Figure 5.13 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 {
  if (original.ownerScope = #classifier) {
    migrated.isStatic = true;
  }
}
migrate Operation {
  if (original.ownerScope = #classifier) {
    migrated.isStatic = true;
  }
```



(a) Original metamodel.

(b) Evolved metamodel.

Figure 5.13: Exemplar UML metamodel evolution

```
23 }
24
25 migrate AssociationEnd to Property {
26   if (original.isNavigable) {
27     original.association.equivalent().navigableEnds.add(migrated)
28   }
29 }
```

Listing 5.21: 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 `Associati-`

`onEnds` are instead stored in `Class#attributes` (line 11).

Comparison

Table 5.4.2 illustrates several characterising differences between Flock and the related approaches presented in Section 5.3.1. Due to its conservative copying algorithm, Flock is the only approach to provide both automatic copying and unsetting. Automatic copying is significant for metamodel evolutions with a large number of unchanging features.

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 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 in Chapter 6.

5.5 Chapter Summary

The solutions 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 from scratch. 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.

	Automatic copy	**Automatic unset**	**Modelling technologies**
Ecore2Ecore	✓	✗	XMI
ATL	✗	✓	EMF, MDR, KM3, XML
COPE	✓	✗	EMF
Flock	✓	✓	EMF, MDR, XML, Z

Table 5.1: Properties of model migration approaches

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[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.

[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.