Chapter 5

Design and Implementation

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

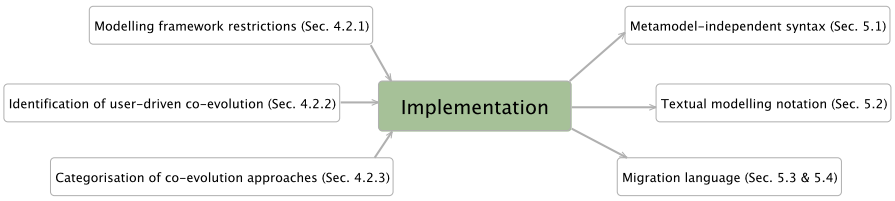


Figure 5.1: Implementation chapter overview.

The structures presented in this chapter are interoperable as shown in Figure 5.2. In particular, the modelling framework extensions provided by the metamodel-independent syntax are used to provide conformance checking for the textual modelling notation, and to enable partial migration for the model migration language. The structures were separated to facilitate re-use of the conformance checking services provided by the metamodel-independent syntax. Table 5 shows the relationship between the proposed structures and the thesis requirements (Section 4.3).

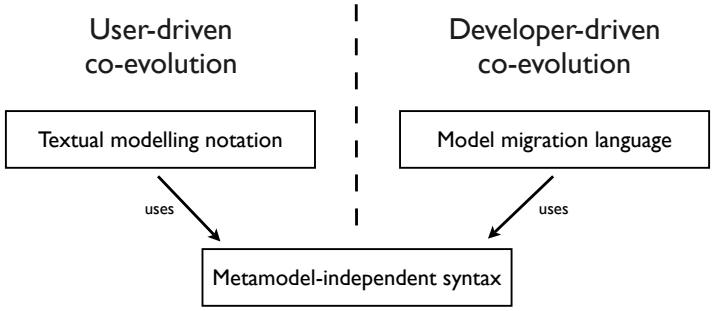


Figure 5.2: The relationships between the proposed structures

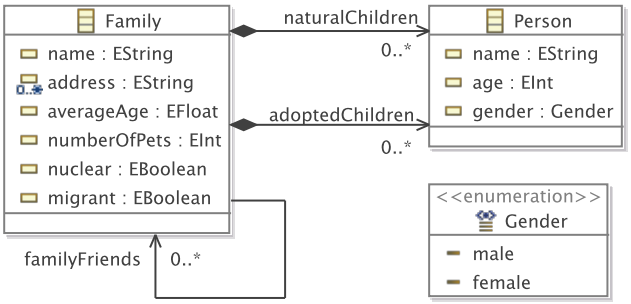
Structure (Section)	**Requirement**
Metamodel-independent syntax (5.1)	This thesis must investigate the extension of existing modelling frameworks to support the loading of non-conformant models and conformance checking of models against other metamodels.
Textual Modelling Notation (5.2)	This thesis 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.
Model Migration Language (5.3)	This thesis must compare and evaluate existing languages for specifying model migration strategies.
Model Migration Language (5.4)	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.

Table 5.1: The relationship between the thesis requirements and the proposed structures.

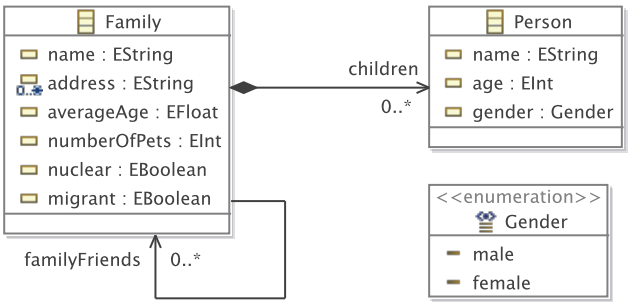
5.1 Metamodel-Independent Syntax

Section 4.2.1 discussed the way in which modelling frameworks implicitly enforce conformance, and hence prevent the loading of non-conformant models. Additionally, modelling frameworks provide little support for checking the conformance of a model with other versions of a metamodel, which is potentially useful during metamodel installation. In Section 4.3, these concerns lead to the identification of the following requirement: *This thesis must investigate the extension of existing modelling frameworks to support the loading of non-conformant models and conformance checking of models against other metamodels.*

This section describes the way in which existing modelling frameworks load and store models using metamodel-specific binding mechanisms, proposes an alternative binding mechanism using a metamodel-independent syntax, and demonstrates how this facilitates automatic consistency checking. The work presented in this section has been published in [Rose *et al.* 2009a].



(a) Original metamodel.



(b) Evolved metamodel.

Figure 5.3: Evolution of a families metamodel, based on the metamodel in [OMG 2004].

5.1.1 Metamodel Evolution Example: Families

This section uses the example of metamodel evolution in Figure 5.3. In Figure 5.3(a), `naturalChildren` and `adoptedChildren` are modelled as separate features, and, in Figure 5.3(b), they are modelled as a single feature, `children`.

Models that specify values for the `naturalChildren` or `adoptedChildren` features do not conform to the evolved metamodel. For example, the model in Figure 5.4 represents a `Family` comprising two `Persons`, conforms to the original metamodel, and does not conform to the evolved metamodel. Using the families metamodel and model, the sequel explains why existing modelling frameworks cannot be used to load non-conformant models.

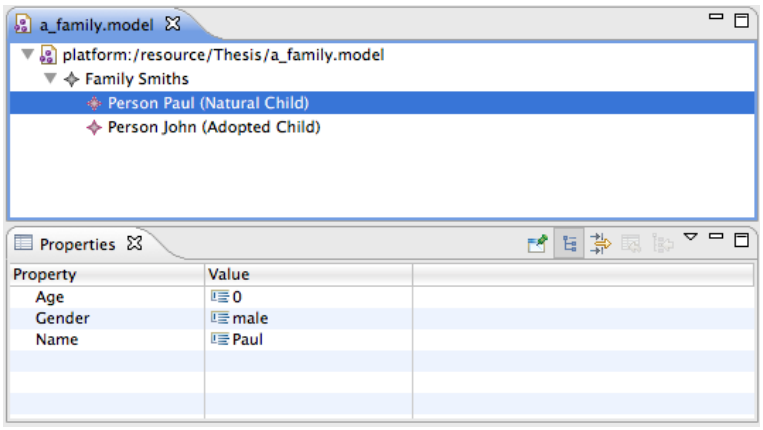


Figure 5.4: A family model, which conforms to the metamodel in Figure 5.3(a)

5.1.2 Binding to a Specific Metamodel

To load a model, existing modelling frameworks construct objects in the underlying programming language in a process termed *binding* (Section 4.2.1). The metamodel defines the way in which model elements will be bound, and binding is strongly-typed. Figure 5.5 illustrates the results of binding the family model in Figure 5.4 to the original families metamodel in Figure 5.3(a). The objects in Figure 5.5 instantiate types that are defined in the metamodel, such as `Family` and `Person`. In other words, binding results in a *metamodel-specific* representation of the model.

Metamodel-specific binding fails for non-conformant models. For example, attempting to bind the family model in Figure 5.4 to the evolved families fails because the model uses `naturalChildren` and `adoptedChildren` features for the type `Family`, and these features are not defined by the metamodel in Figure 5.3(b).

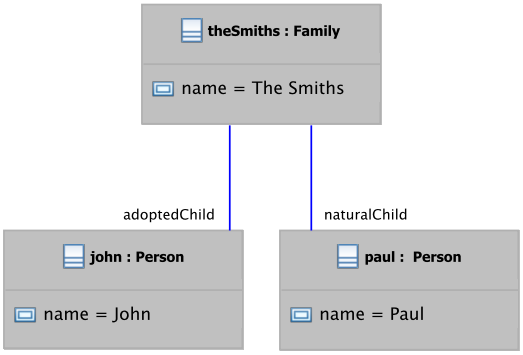


Figure 5.5: Objects resulting from the binding of a conformant model

Because non-conformant models cannot be loaded, model migration must be performed by editing the underlying storage representation, which can be error-prone and tedious (Section 4.2.2). The sequel discusses potential solutions for loading non-conformant models.

5.1.3 Potential Solutions for Loading Non-Conformant Models

Two potential approaches to binding (and hence loading) non-conformant models have been considered and are now discussed. The benefits and drawbacks of each approach have been compared, which resulted in the selection of the second approach, binding to a metamodel-independent syntax.

Store metamodel history

Presently, modelling frameworks are used to store only the latest version of a metamodel, and hence binding fails for models that conform to a previous version of the metamodel. If modelling frameworks could access old versions of a metamodel, models that do not conform to the current version of the metamodel could be loaded by binding to a previous version of the metamodel.

A metamodel-independent syntax

Models can always be successfully bound to a *metamodel-independent* representation, such as the one shown in Figure 5.6. Binding each model element results in the instantiation of a metamodel-independent type (`Object` in Figure 5.6) rather than of types defined in a specific metamodel, such as `Family` or `Person`. Hence, binding is independent of the types defined in metamodels, and will succeed for non-conformant models.

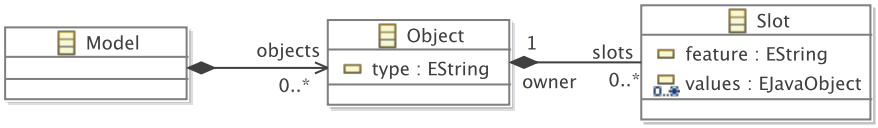


Figure 5.6: A minimal generic metamodel for MOF in Ecore, based on [OMG 2008a] and taken from [Rose *et al.* 2009a].

Benefits and drawbacks of the potential solutions

The two potential solutions for loading non-conformant models have different benefits and drawbacks, which are now discussed. Storing metamodel histories would use the binding and conformance checking services provided by existing modelling frameworks, and therefore require less implementation effort than a metamodel-independent syntax, which would require bespoke binding and conformance checking services. Furthermore, structures for managing metamodel histories might be integrated with existing approaches to managing co-evolution, such as metamodel differencing approaches (Sections 4.2.3), for switching between different versions of a MDE workflow.

Storing metamodel histories relies on the metamodel developer to enable model migration: if the metamodel developer does not provide a metamodel that contains historical data, then binding will fail for non-conformant models. Conversely, models can be bound to a metamodel-independent syntax irrespective of the actions of the metamodel developer.

A metamodel-independent syntax has been chosen because it makes fewer assumptions of the metamodel developer, and hence facilitates user-driven as well as developer-driven co-evolution.

5.1.4 Proposed Solution: A Metamodel-Independent Syntax

This section discusses the design and implementation of a metamodel-independent syntax, and of the binding and conformance checking services that are used to load non-conformant models. As discussed below, the design of the metamodel-independent syntax and conformance checking service is inspired by [OMG 2007a] and [Paige *et al.* 2007], respectively. As such, the primary contribution of this section is the implementation and integration of the syntax and services with EMF. In addition, the syntax and services have been designed to be re-usable, and hence have been used to simplify the implementation of a textual modelling notation (Section 5.2) and a model migration language (Section 5.4).

Design

A high-level design for the way in which the metamodel-independent syntax, binding service and conformance checking service load models is shown in

Figure 5.7. The **binding service** parses XMI (the canonical storage representation of models, Section 2.1.3) and produces a model that conforms to the **metamodel-independent syntax**. The **conformance checking service** is used to explicitly check the conformance of a model conforming to the metamodel-independent syntax.

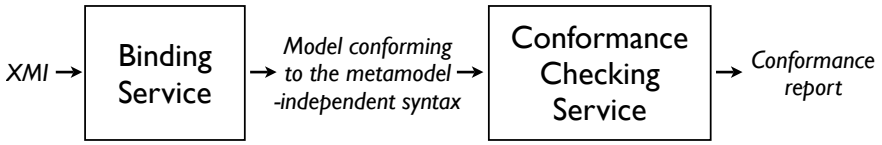


Figure 5.7: Loading models with the metamodel-independent syntax

Binding and conformance checking were split into separate services to facilitate re-use. For example, the textual modelling notation in Section 5.2 re-uses the metamodel-independent syntax and conformance checking service, in conjunction with a different binding service.

Metamodel-independent syntax The metamodel-independent syntax is used to represent a model without instantiating types defined by its metamodel. Its design was inspired by the metamodel for UML 2 [OMG 2007a] object diagrams, which describes objects in a generic, class-independent manner. UML 2 object diagrams are specified in terms of an abstract syntax (comprising, for example, `InstanceSpecification` and `Link` classes) and a concrete syntax (comprising, for example, boxes and lines). The metamodel-independent syntax proposed here is abstract. It is not used directly by metamodel developers or users and hence a concrete syntax was not required.

Abstract syntax is typically represented as a metamodel (Section 2.1.2). The metamodel in Figure 5.6 was used as an initial design for the metamodel-independent syntax, which contains a class for each type in the MOF metamodel that is instantiated in a model. In other words, `Objects` are used to represent each element of a model, and the `type` attribute is used to indicate the name of the metaclass that the `Object` intends to instantiate. Similarly, `Slots` are used to represent values in the model, and the `feature` attribute indicates the metafeature that the `Slot` intends to instantiate. The metamodel was designed to capture the information needed to perform conformance checking (described below), and implementing the conformance checking service led to a refactored metamodel, which is presented in the sequel.

COPE (Section 4.2.3) is also built atop a metamodel-independent syntax. However, the metamodel-independent syntaxes used by COPE and proposed here were developed independently, and both were first published in 2008 (in [Rose *et al.* 2008a, Herrmannsdoerfer *et al.* 2008b]).

Metamodel-independent binding service Binding a textual representation of a model to a metamodel is a model-to-text (M2T) transformation. The metamodel-independent binding service is a M2T transformation that consumes XMI and produces a model conforming to the metamodel-independent syntax. The transformation iterates over each tag in the XMI, and creates instances of `Object` and `Slot`. For example, when encountering a tag that represent a model element, the transformation performs the steps in Figure 5.8.

Applying the metamodel-independent binding service to the families model (Figure 5.4) produces three instances of `Object`, illustrated as a UML object diagram in Figure 5.9. For clarity, instances of `Object` are shaded, and instances of `Slot` are unshaded. The first `Object` represents the `Family` model element and has three slots. Two of the slots are used to reference the `Person` model elements via the `naturalChildren` and `adoptedChildren` references.

Conformance checking service Conformance is a type of inter-model consistency, between a model and its metamodel (Section 2.1.2), and, in MDE, inter-model consistency is often validated using a set of constraints (Section 2.1.4). Furthermore, [Paige *et al.* 2007] demonstrates that conformance can be specified as a set of constraints between a model and its metamodel. As such, the conformance checking service has been designed as the set of constraints between models and metamodels in Figure 5.11.

The conformance checking service must be interoperable with the metamodel-independent syntax and, hence, the constraints are specified in terms of `Objects` and `Slots`. Clearly, to check conformance the constraints must refer to a (specific) metamodel, and the constraints are also specified in terms of concepts from the MOF metamodelling language (Section 2.1.3), such as `Class` and `Property`. Figure 5.10 shows a minimal version of the MOF metamodel.

After binding to the metamodel-independent syntax, the conformance of a model can be checked against any specific metamodel. To illustrate the value of the conformance checking service, consider again the metamodel evolution in Figure 5.3 and the bound model in Figure 5.9. For the evolved metamodel (Figure 5.3(b)), conformance checking for the model element representing the `Family` would fail. As illustrated in Figure 5.9, the `Family Object` defines slots for features named `naturalChildren` and `adoptedChildren`, which are not defined the metaclass `Family` in Figure 5.3(b). Specifically, the model element representing the `Family` does not satisfy conformance constraint 3, which states: *each `Slot`'s `feature` must be the name of a metamodel `Property`. That `Property` must belong to the `Slot`'s `owner`'s `type`.*

Reference implementation in Java, EMF and Epsilon

Reference implementations of the three components were constructed with Java, EMF and Epsilon (Section 2.3). The way in which each component was

1. Constructs an instance of `Object`, `o`.
2. For each attribute of the tag:
   - Creates an instance of `Slot`, `s`.
   - Sets `s.feature` to the name of the attribute.
   - Sets `s.value` to the value of the attribute.
   - Adds `s` to `o.slots`.
3. For each child tag:
   - Creates an instance of `Slot`, `s`.
   - Sets `s.feature` to the name of the child tag.
   - Recursively constructs an instance of `Object`, c.
   - Sets `s.value` to c.
   - Adds `s` to `o.slots`.

Figure 5.8: Pseudo code for binding XMI tags to `Objects`.

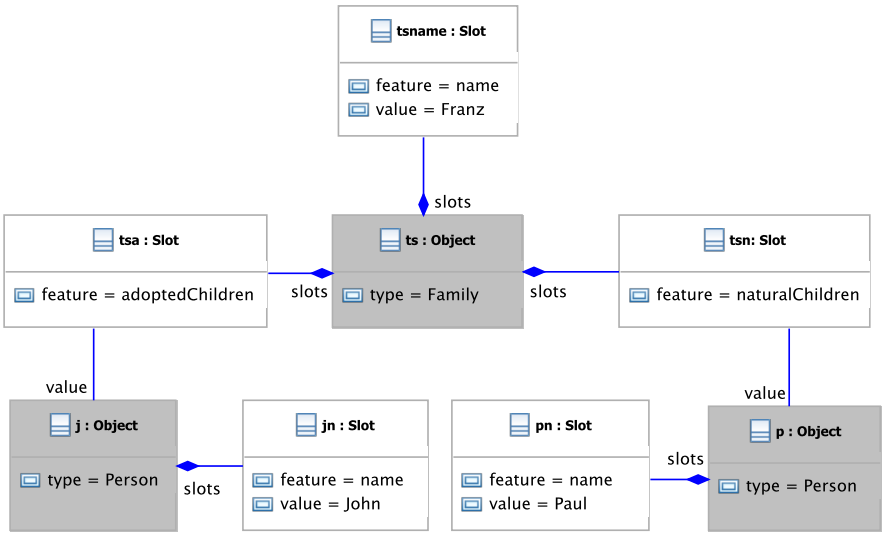


Figure 5.9: Result of binding the families model with the metamodel-independent syntax

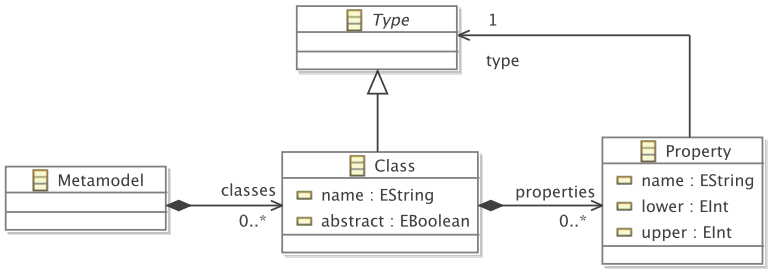


Figure 5.10: Minimal MOF metamodel, based on [OMG 2008a].

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 `Property` of its `type`.
3. Each `Slot`'s `feature` must be the name of a metamodel `Property`. That `Property` must belong to the `Slot`'s `owner`'s `type`.
4. Each `Slot` must be multiplicity-compatible with its `Property`. More specifically, each `Slot` must contain at least as many values as its `Property`'s `lower` bound, and at most as many values as its `Property`'s `upper` bound.
5. Each `Slot` must be type-compatible with its `Property`. (The way in which type-compatibility is checked depends on the way in which the modelling framework is implemented).

Figure 5.11: The constraints of the conformance checking service.

implemented is now discussed.

Metamodel-independent syntax Ecore, the metamodelling language of EMF, was used to implement the metamodel-independent syntax. The final metamodel is shown in Figure 5.12, which differs slightly to the initial design (Figure 5.6). Specifically, `Slot` is abstract, has a generic type (`T`), and is the superclass of `AttributeSlot`, `ReferenceSlot` and `ContainmentSlot`. These changes simplified the implementation of the (abstract) `typeCompatibileWith` method, which is used by the conformance checking service, and returns `true` if and only if every element of the `values` attribute is type compatible with the `EClassifier` parameter (a metamodel type).

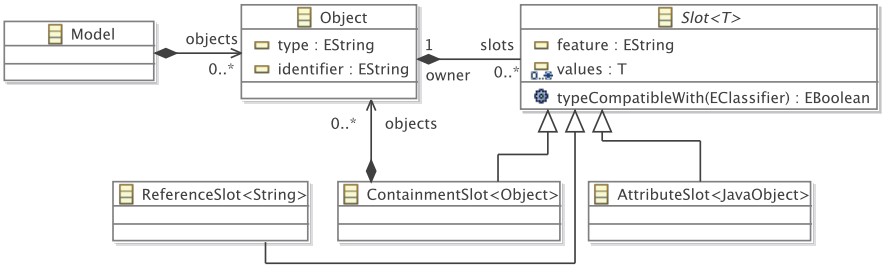


Figure 5.12: Implemented version of the metamodel-independent syntax, in Ecore

Binding service A text-to-model (T2M) transformation language (Section 2.1.4) could have been used to implement the binding service. However, in 2008 the Eclipse Modeling Project[1] did not provide a standard T2M language and using a T2M language that was not part of the Eclipse Modeling Project would have complicated installation of the service for users.

Instead, the binding service has been implemented by constructing in Java an XMI parser that emits objects conforming to the metamodel-independent syntax. Listing 5.1 illustrates the way in which XMI attributes are parsed. The `processAttributes` method is called to generate instances of `AttributeSlot` from the metamodel-independent syntax. For each attribute in an XMI tag, the body of the loop is executed. If the attribute is not XMI metadata such as type information (line 4), the name and value of the attribute (lines 5 and 6) are extracted from the XMI, and used to add the value to an `AttributeSlot` with feature equal to the name of the attribute (line 8). Constructing `Objects` and `Slots` is the responsibility of the `generator` object, which is an instance variable of the parser.

[1] `http://www.eclipse.org/modeling/`

```
1  private void processAttributes(Attributes atts) {
2   for (int index = 0; index < atts.getLength(); index++) {
3
4    if (!attributeIsMetadata(atts.getQName(index))) {
5     final String feature = atts.getLocalName(index);
6     final String value = atts.getValue(index);
7
8     generator.addAttributeValue(feature, value);
9    }
10  }
11 }
```

Listing 5.1: Parsing XMI attributes (in Java)

Conformance checking service The conformance constraints (Figure 5.11) were implemented with EVL (Section 2.1.4), a language tailored for (inter-)model verification and hence suitable for rapid prototyping of consistency constraints. Listing 5.2 shows the EVL constraint that checks whether each `Object`'s `type` is a non-abstract class (constraint 1 in Figure 5.11). The `check` part (line 3) verifies that a particular `Object` (referenced via the `self` keyword) refers to a metamodel type that is not abstract. When the check fails, the message (line 4) is automatically added to a set of unsatisfied constraints. The `toClass` operation (lines 8-10) is used to determine the metamodel class (an instance of `EClass`) to which the `type` attribute (a String) of an `Object` refers. The conformance checking service returns a report of unsatisfied constraints.

```
1  context Object {
2   constraint ClassMustNotBeAbstract {
3    check: not self.toClass().isAbstract()
4    message: 'Cannot instantiate the abstract class: ' + self.type
5   }
6  }
7
8  operation Object toClass() : EClass {
9   return Metamodel!EClass.all.selectOne(c|c.name == self.type);
10 }
```

Listing 5.2: A constraint (in EVL) to check that only concrete metamodel types are instantiated.

Type-compatibility has been implemented by delegating to the type-checking methods provided by EMF. The EVL constraints call the `isTypeCompatibleWith` method on the `Slot` class. Each subclass of `Slot` provides an implementation of `isTypeCompatibleWith`, which delegates to EMF to perform type-checking.

5.1.5 Structures Built atop the Metamodel-Independent Syntax

There are many potential uses for the metamodel-independent syntax described in this section. Section 5.2 describes a textual modelling notation integrated with the metamodel-independent syntax to achieve live conformance checking. The migration language presented in Section 5.4 can be used with the metamodel independent syntax to perform partial migration.

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

To enable conformance checking as part of metamodel installation in EMF, the metamodel-independent syntax has been integrated with Concordance in [Rose *et al.* 2010c]. The work was conducted outside of the scope of the thesis, and is now summarised to indicate the usefulness of the metamodel-independent syntax for supporting the automation of co-evolution activities. Concordance provides a mechanism for resolving inter-model references (such as those between models and their metamodels). Without Concordance, determining the the instances of a metamodel is possible only by checking every model in the workspace. Integrating Concordance and the metamodel-independent syntax resulted in a service, which Epsilon (Section 2.3.2) executes after the installation of a metamodel to identify the models that are affected by the metamodel changes. All models that conform to the old version of the metamodel are checked for conformance with the new metamodel. As such, conformance checking occurs automatically and immediately after metamodel installation. Conformance problems are detected and reported immediately, rather than when an affected model is next used.

Summary

Modelling frameworks implicitly enforce conformance, which presents challenges for managing co-evolution. In particular, detecting and reconciling conformance problems involves managing non-conformant models, which cannot be loaded by modelling frameworks and hence cannot be used with model editors or model management operations. The metamodel-independent syntax proposed in this section enables modelling frameworks to load non-conformant

models, and has has been integrated with Concordance [Rose *et al.* 2010c] to facilitate the reporting of conformance problems during metamodel installation. The metamodel-independent syntax, binding service and conformance checking service underpin the implementation of the textual modelling notation presented in the sequel. The benefits and drawbacks of the metamodel-independent syntax in the context of user-driven co-evolution are explored in Chapter 6.

5.2 Textual Modelling Notation

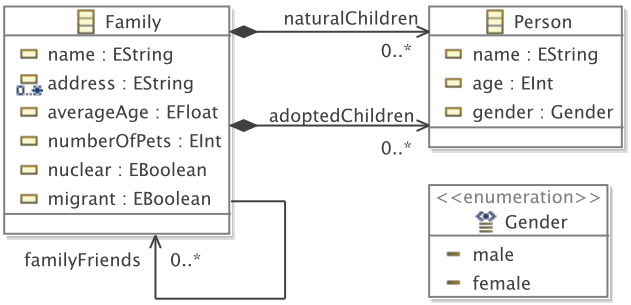
The analysis of co-evolution examples in Chapter 4 highlighted two ways in which co-evolution is managed. In *developer-driven* co-evolution, migration is specified by the metamodel developer in an executable format; while in *user-driven co-evolution* migration is specified by the metamodel developer in prose or not at all. Performing user-driven co-evolution with modelling frameworks presents two key challenges that have not been explored by existing research. Firstly, user-driven co-evolution often 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 led 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.

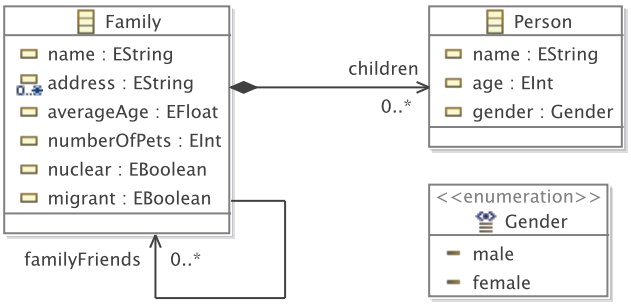
5.2.1 Model Migration with XMI

The co-evolution example from Section 5.1 is now used to illustrate the way in which model migration is performed by editing the underlying storage representation of a model, such as XMI (Section 2.1.3). Consider again the evolution of the families metamodel (Figure **??**) and a model conforming to the original metamodel (Figure **??**).

The model in Figure **??** does not conform to the evolved metamodel (because it uses the `naturalChildren` and `adoptedChildren` features, which are not defined for `Person`), and hence cannot be loaded by the modelling framework. Migration might be achieved by editing the underlying



(a) Original metamodel.



(b) Evolved metamodel.

Figure 5.13: Evolution of a families metamodel, based on the metamodel in [OMG 2004].

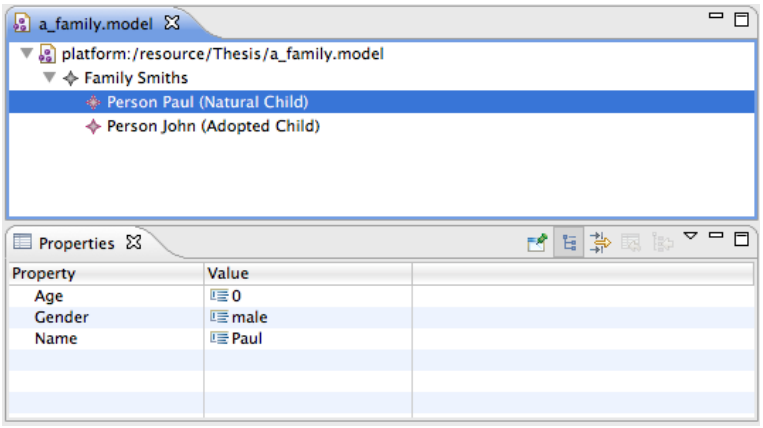


Figure 5.14: A family model, which conforms to the metamodel in Figure **??**

storage representation directly (i.e. manually manipulating XMI). Listing **??** shows the XMI for the model in Figure **??**.

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

Listing 5.3: XMI for the family model in Figure **??**

5.2.2 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: [Steel & Raymond 2001] describe the Distributed Systems Technology Centre's TokTok project (an implementation of the HUTN specification), which is now inactive (and the source code has vanished), whilst the implementation of HUTN described in [Muller & Hassenforder 2005] has been abandoned in favour of Sintaks[2], which operates on domain-specific concrete syntax.

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

Like the generic metamodel presented in Section 5.1, HUTN is a metamodel-independent syntax for MOF. However, the HUTN specification focuses on concrete syntax, whereas the metamodel-independent syntax presented in Section 5.1 focuses on abstract syntax. In this section, the key features of HUTN are introduced, and the sequel introduces a reference implementation of HUTN. To illustrate the notation, the MOF-based metamodel of families in Figure 5.13 is used. The `nuclear` attribute on the `Family` class is used to indicate that the family "comprises only a father, a mother, and children." [Merriam-Webster 2010].

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

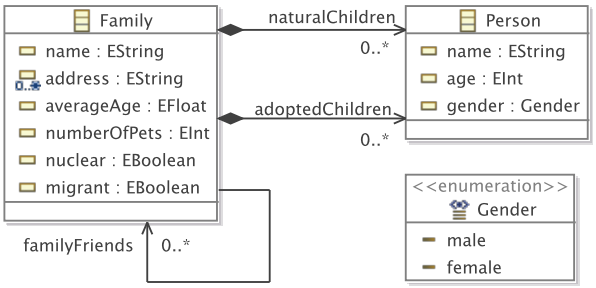


Figure 5.15: Exemplar families metamodel

Basic Notation

Listing 5.3 shows the construction of an *object* in HUTN, here an instance of the Family class from Figure 5.13. Line 1 specifies the package containing the classes to be constructed (`FamilyPackage`) and a corresponding identifier (`families`), used in fully-qualified references to objects (Section 5.2.1). Line 2 names the class (`Family`) and gives an identifier for the object (`The Smiths`). Lines 3 to 7 define *attribute values*; in each case, the data value is assigned to the attribute with the specified name. The encoding of the value depends on its type: strings are delimited by any form of quotation mark; multi-valued attributes use comma separators, etc.

The metamodel in Figure 5.13 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 5.4. A simple reference can be specified using the type and identifier of the referred object, as shown in Listing 5.5. 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 5.4: Specifying attributes with HUTN.

```
1 FamilyPackage "families" {
2    Family "The Smiths" {
```

```
3        naturalChildren: Person "John" { name: "John" },
4                         Person "Jo" { gender: female }
5     }
6  }
```

Listing 5.5: Specifying a containment reference with HUTN.

```
1  FamilyPackage "families" {
2     Family "The Smiths" {
3        familyFriends: Family "The Does"
4     }
5     Family "The Does" {}
6  }
```

Listing 5.6: 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 5.6 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 5.7: 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).

```
1  FamilyPackage "families" {
2     Family "The Smiths" {}
3  }
4  VehiclePackage "vehicles" {
5     Vehicle "The Smiths' Car" {
6        owner: FamilyPackage.Family "families"."The Smiths"
7     }
8  }
```

Listing 5.8: 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 5.13. Among other concepts, this package introduces a Vehicle class, which defines an owner reference of type Family. Listing 5.7 illustrates the way in which the owner feature can be populated. Note that the fully-qualified form of the class utilises the names of elements of the metamodel, while the fully-qualified form of the object utilises only HUTN identifiers defined in the current document.

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

Alternative Reference Syntax

In addition to the syntax defined in Listings 5.4 and 5.5, the value of references may be specified independently of the object definitions. For example, Listing 5.8 demonstrates this alternate syntax by defining The Does as friends with both The Smiths and The Bloggs.

```
1  FamilyPackage "families" {
2     Family "The Smiths" {}
3     Family "The Does" {}
4     Family "The Bloggs" {}
5
6     familyFriends {
7        "The Does" "The Smiths"
8        "The Does" "The Bloggs"
9     }
10 }
```

Listing 5.9: Using a reference block in HUTN.

Listing 5.9 illustrates a further alternative syntax for references, which employs an infix notation.

```
1  FamilyPackage "families" {
2     Family "The Smiths" {}
3     Family "The Does" {}
4     Family "The Bloggs" {}
5
6     Family "The Smiths" familyFriends Family "The Does";
7     Family "The Smiths" familyFriends Family "The Bloggs";
8  }
```

Listing 5.10: Using an infix reference in HUTN.

The reference block (Listing 5.8) and infix (Listing 5.9) notations are syntactic variations on – and have identical semantics to – the reference notation shown in Listings 5.4 and 5.5.

Customisation via Configuration

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

5.2.3 Epsilon HUTN

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

Implementation of Epsilon HUTN

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

domain-specific languages are tightly integrated and inter-operable, making it feasible to chain model management operations together to implement Epsilon HUTN.

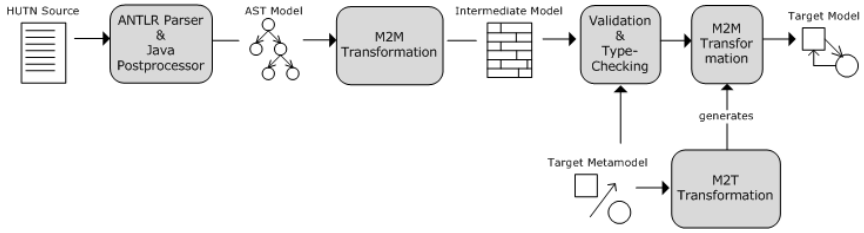


Figure 5.16: The architecture of Epsilon HUTN.

Figure 5.14 outlines the workflow through Epsilon HUTN, from HUTN source text to instantiated target model. The HUTN model specification is parsed to an abstract syntax tree using a HUTN parser specified in ANTLR [Parr 2007]. From this, a Java postprocessor is used to construct an instance of a simple AST metamodel (which comprises two meta-classes, Tree and Node). Using ETL, M2M transformations are then applied to produce an intermediate model, which is an instance of the generic metamodel discussed in Section 5.1. Finally, a M2T transformation on the target metamodel, specified in EGL, produces a further M2M transformation, which consumes the intermediate model and produces the target model.

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

Using two M2M transformation stages with the (extended) generic metamodel as an intermediary has two advantages. Firstly, the form of the AST metamodel is not suited to a one-step transformation. There is a mismatch between the features of the AST metamodel and the needs of the target model – for example, between the Node class in the AST metamodel and classes in the target metamodel. If a one-step transformation were used, each transformation rule would need a lengthly guard statement, which is hard to understand and verify. Secondly, Section 5.1 discussed a mechanism for binding XMI to the generic metamodel, which can be used in conjunction with the latter half of the Epsilon HUTN workflow (Figure 5.14) to generate HUTN from XMI. This process is discussed further in Section 5.2.3.

Throughout the remainder of this section, instances of the generic metamodel producing during the execution of the HUTN workflow are termed an *intermediate model*. The two M2M transformations are now discussed in

depth, along with a model validation phase which is performed prior to the second transformation.

AST Model to Intermediate Model Epsilon HUTN uses ETL [Kolovos *et al.* 2008b] for specifying M2M transformation. One of the transformation rules from Epsilon HUTN is shown in Listing 5.10. 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 5.11: Transforming Nodes to PackageObjects with ETL.

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.

```
1  context ClassObject {
2     constraint IdentifiersMustBeUnique {
3        guard: self.id.isDefined()
```

```
4         check: ClassObject.all
5                 .select(c|c.id = self.id).size() = 1;
6         message: 'Duplicate identifier: ' + self.id
7     }
8 }
```

Listing 5.12: A constraint (in EVL) to check that all identifiers are unique.

Epsilon HUTN uses EVL [Kolovos *et al.* 2009] 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 5.11 specifies the constraint that every HUTN class object has a unique identifier.

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

Intermediate Model to Target Model When the intermediate model conforms to the target metamodel, the intermediate model can be transformed to an instance of the target metamodel. In generating the target model from the intermediate model (Figure 5.14), the transformation uses information from the target metamodel, such as the names of classes and features. A typical approach to this category of problem is to use a higher-order transformation on the target metamodel to generate the desired transformation. Epsilon HUTN uses a different approach: the transformation to the target model is produced by executing a M2T transformation on the target metamodel. EGL [Rose *et al.* 2008b] is a template-based M2T language. `[% %]` tag pairs are used to denote dynamic sections, which may produce text when executed. Any code not enclosed in a `[% %]` tag pair is included verbatim in the generated text.

Listing 5.12 shows part of the M2T transformation used by HUTN. The M2T transformation generates a M2M transformation which specifies the way in which the intermediate model is transformed to the target model. The loop beginning on line 1 of Listing 5.12 iterates over each meta-class in the target metamodel, producing a M2M transformation rule. The generated transformation rule consumes a class objects in the intermediate model and produces an element of the target model. The guard of the generated transformation rule (line 6) ensures that only class objects with a type equal to the current meta-class are transformed by the generated rule. To generate the body of the rule, the M2T transformation iterates over each structural feature of the current meta-class, and generates appropriate transformation code for populating the values of each structural feature from the slots on the class object in the intermediate model. The part of the M2T transformation that generates the body of M2M transformation rule is omitted in Listing 5.12 because

it contains a large amount of code for interacting with EMF, which is not relevant to this discussion.

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

Listing 5.13: Part of the M2T transformation (in EGL) for generating the intermediate model to target model transformation (in ETL).

For example, executing the M2T transformation in Listing 5.12 on the Families metamodel (Figure 5.13) generates the two M2M transformation rules shown in Listing 5.13. The rules produce instances of Family and Person from instances of ClassObject in the intermediate model. The body of each rule copies the values from the slots of the ClassObject to the Family or Person in the target model. Lines 7-9, for example, copy the value of the name slot (if one is specified) to the target Family.

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

```
24      guard: o.type = 'Person'
25
26      if (o.hasSlot('name')) {
27        t.name := o.findSlot('name').values.first;
28      }
29
30      -- remainder of body omitted
31    }
```

Listing 5.14: The M2M transformation generated for the Families metamodel

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

5.2.4 Migration with HUTN

Used in combination with the metamodel-independent syntax presented in Section 5.1, Epsilon HUTN facilitates user-driven co-evolution. To this end, Epsilon HUTN provides development tools (menu items and user-interface wizards) to realise the workflow shown in Figure 5.15, which provides an alternative to the user-driven co-evolution workflow observed in Section 4.2.2. (The workflow in Figure 5.15 assumes a graphical model editor, such as those generated by GMF, but any editor built atop EMF will exhibit the same behaviour). First, the user attempts to load a model in the graphical editor. If the model is non-conformant and cannot be loaded, the user clicks the "Generate HUTN" menu item, and the model is loaded with the metamodel-independent syntax and then a HUTN representation of the model is generated by Epsilon HUTN. The generated HUTN is presented in an editor that automatically reports conformance problems using the metamodel-independent syntax. The user edits the HUTN to reconcile conformance problems, and the conformance report is automatically updated as the user edits the model. When the conformance problems are fixed, XMI for the conformant model is automatically generated by Epsilon HUTN, and migration is complete. The model can then be loaded in the graphical editor.

To demonstrate the way in which HUTN can be used to perform migration, the exemplar XMI shown in Listing **??** is represented using HUTN in Listing 5.14. Recall that the XMI describes three Persons, Franz, Julie and Hermann. Julie and Hermann are the mother and father of Franz.

```
1  Persons "kafkas" {
2      Person "Franz" { name: "Franz" }
3      Person "Julie" { name: "Julie" }
```

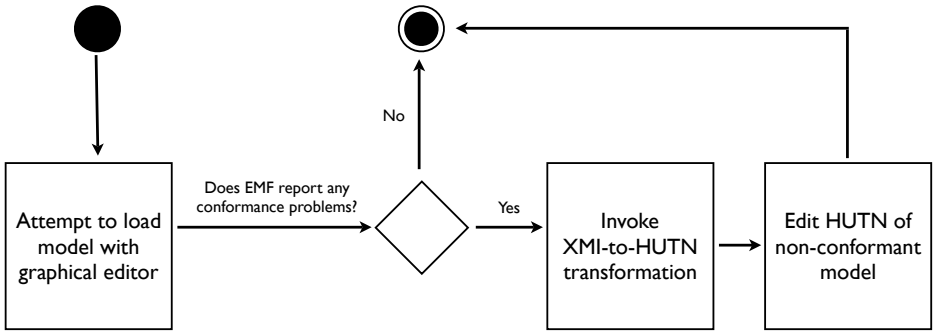


Figure 5.17: User-driven co-evolution with dedicated structures

```
4     Person "Hermann" { name: "Hermann" }
5
6     Person "Franz" mother Person "Julie";
7     Person "Franz" father Person "Hermann";
8  }
```

Listing 5.15: HUTN for people with mothers and fathers.

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

If the Persons metamodel now evolves such that mother and father are merged to form a parents reference, Epsilon HUTN reports conformance problems on the HUTN document, as illustrated by the screenshot in Figure 5.16.

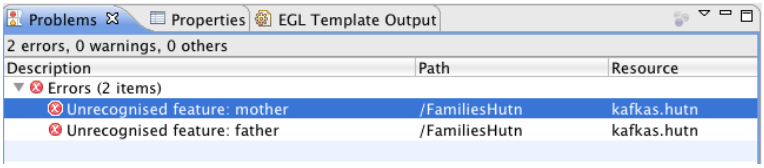


Figure 5.18: Conformance problem reporting in Epsilon HUTN.

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 5.15 shows a HUTN document that conforms to the metamodel defining parents rather than mother and father.

```
1  Persons "kafkas" {
2     Person "Franz" { name: "Franz" }
3     Person "Julie" { name: "Julie" }
4     Person "Hermann" { name: "Hermann" }
```

```
5
6     Person "Franz" parents Person "Julie";
7     Person "Franz" parents Person "Hermann";
8  }
```

Listing 5.16: HUTN for people with parents.

5.2.5 Limitations

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

Generic vs specific concrete syntax

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

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

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

The above expression is written using a domain-specific syntax. In HUTN, the specification of these behaviours is less concise. For example, Listing 5.16 gives the HUTN syntax for failure behaviour (5.1), above.

```
1  Behaviour {
2     lhs: Tuple {
3        contents: IdentifierSet { contents: Wildcard {} },
4                  IdentifierSet { contents: Wildcard {} }
5     }
6
7     rhs: Tuple {
8        contents: IdentifierSet { contents: Fault "late" {} }
9     }
10 }
```

Listing 5.17: Failure behaviour specified in HUTN.

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

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

Optimised XMI omits type information

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

5.2.6 Summary

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

5.3 Analysis of Languages used for Migration

In contrast to the previous section, this section focuses on *developer-driven* co-evolution, in which migration is specified as a program that metamodel users execute to migrate their models. Section 4.2.3 discussed existing approaches to model migration, highlighting variation in the languages used for specifying migration strategies. In this section, the results of comparing migration strategy languages are described, using a new example of metamodel evolution (Section 5.3.1). From the comparison, requirements for a domain-specific language for specifying and executing model migration strategies are derived (Section 5.3.3). The sequel describes an implementation of a model migration

language based on the analysis presented here. The work described in this section has been published in [Rose *et al.* 2010f].

5.3.1 Co-Evolution Example

This section uses the Petri net metamodel evolution to compare model migration languages. The example is also used in co-evolution literature [Cicchetti *et al.* 2008, Garcés *et al.* 2009, Wachsmuth 2007].

In Figure 5.17(a), a Petri `Net` is defined to comprise `Places` and `Transitions`. A `Place` has any number of `src` or `dst` `Transitions`. Similarly, a `Transition` has at least one `src` and `dst` `Place`. The metamodel is to be evolved to support weighted connections between `Places` and `Transitions` and between `Transitions` and `Places`, as shown in Figure 5.17(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 conform 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`.

   Set s as the `src` of arc.

   Set t as the `dst` of arc.

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

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

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

   Set t as the `src` of arc.

   Set d as the `dst` of arc.

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

2. And nothing else changes.

5.3.2 Existing Model Migration Languages

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 M2M Transformation

Model migration can be specified using M2M transformation. For example, the Petri net migration has been specified in the Atlas Transformation Language (ATL) [Jouault & Kurtev 2005]. This is reproduced in Listing 5.17. Rules for migrating `Places` and `TPArcs` have been omitted for brevity, but are similar to the `Nets` and `PTArcs` rules.

Model transformation in ATL is specified using `rules`, which transform source model elements (specified using the `from` keyword) to target model elements (specified using `to` keyword). For example, the `Nets` rule on line 1 of Listing 5.17 transforms an instance of `Net` from the original (source) model to an instance of `Net` in the evolved (target) model. The source model element (the variable `o` in the `Net` rule) is used to populate the target model element (the variable `m`). ATL allows rules to be specified as *lazy* (not scheduled automatically and applied only when called by other rules).

The `Transitions` rule in Listing 5.17 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`.

```
1  rule Nets {
2    from o : Before!Net
3    to m : After!Net ( places <- o.places, transitions <- o.transitions )
4  }
5
6  rule Transitions {
7    from o : Before!Transition
8    to m : After!Transition (
9       name <- o.name,
10      "in" <- o.src->collect(p | thisModule.PTArcs(p,o)),
11      out <- o.dst->collect(p | thisModule.TPArcs(o,p))
12    )
13 }
14
15 unique lazy rule PTArcs {
16   from place : Before!Place, destination : Before!Transition
17   to ptarcs : After!PTArc (
18      src <- place, dst <- destination, net <- destination.net
19    )
20 }
```

Listing 5.18: Fragment of the Petri nets model migration in ATL, taken from

[Rose *et al.* 2010f]

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. M2M transformation languages typically support new-target transformations. Some M2M transformation languages also support existing-target transformations, but typically require the source and target metamodel to be identical.

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

Manual Specification with a Metamodel Mapping

Model migration can be undertaken using the model loading mechanisms of EMF [Hussey & Paternostro 2006], with a tool that is termed *Ecore2Ecore* here. The default model loading mechanism provided by EMF binds models to their metamodel (Section 4.2.1), and hence cannot be used to load models that have been affected by metamodel evolution. Therefore, Ecore2Ecore 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 Hussey and Paternostro's tool, which can suggest relationships between source and target metamodel elements by comparing names and types. For the Petri nets example, Figure 5.18 shows mappings between the original and evolved metamodels.

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

In Ecore2Ecore, model migration is specified on the XMI representation of the model and requires 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 5.18 shows a fragment of the code used to migrate Petri net models with Ecore2Ecore. 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.

```
1  private Collection<Place> toCollectionOfPlaces
2  (String value, Resource resource) {
3
4    final String[] uriFragments = value.split("␣");
5    final Collection<Place> places = new LinkedList<Place>();
6
7    for (String uriFragment : uriFragments) {
8      final EObject eObject = resource.getEObject(uriFragment);
9      final EClass place = PetriNetsPackage.eINSTANCE.getPlace();
10
11     if (eObject == null || !place.isInstance(eObject))
12       // throw an exception
13
14     places.add((Place)eObject);
15   }
16
17   return places;
18 }
```

Listing 5.19: 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.* 2009a] 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 COPE tracks their application. Once metamodel evolution is complete, a migration strategy can be generated automatically from the record of changes maintained by

COPEs. The migration strategy is distributed along with the updated metamodel, and metamodel users choose when to execute the migration strategy on their models.

To be effective, operator-based approaches must provide a rich yet navigable library of co-evolutionary operators (Section 4.2.3). COPE allows model migration strategies to be specified manually when no co-evolutionary operator is appropriate. COPE employs a fundamentally different approach to M2M transformation and Ecore2Ecore, using an existing-target transformation. As discussed above, existing-target transformations cannot be used for specifying model migration strategies as the source (original) and target (evolved) metamodels differ. However, models can be structured independently of their metamodel using a metamodel-independent syntax (such as the one introduced in Section 5.1).

Listing 5.19 shows the COPE model migration strategy for the Petri net example given above[3]. Most notably, slots for features that no longer exist must be explicitly `unset`. In Listing 5.19, slots are `unset` on four occasions (on lines 2, 9, 18 and 19), once for each feature that is in the original metamodel but not in the evolved metamodel. 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.

```
1  for (transition in petrinets.Transition.allInstances) {
2    for (source in transition.unset('src')) {
3      def arc = petrinets.PTArc.newInstance()
4      arc.src = source
5      arc.dst = transition
6      arc.net = transition.net
7    }
8
9    for (destination in transition.unset('dst')) {
10     def arc = petrinets.TPArc.newInstance()
11     arc.src = transition
12     arc.dst = destination
13     arc.net = transition.net
14   }
15 }
16
17 for (place in petrinets.Place.allInstances) {
18   place.unset('src')
19   place.unset('dst')
20 }
```

Listing 5.20: Petri nets model migration in COPE

[3]In Listing 5.19, some of the concrete syntax has been changed in the interest of readability.

5.3.3 Requirements Identification

Requirements for a domain-specific for model migration were identified from the review of existing languages (Section 5.3.2). The derivation of the requirements is now summarised, by considering two orthogonal concerns: the source-target relationship of the language used for specifying migration strategies and the way in which models are represented during migration.

Source-Target Relationship Requirements

When migration is specified as a new-target transformation, as in ATL (Listing 5.17), model elements that have not been affected by metamodel evolution must be explicitly copied from the original to the migrated model. When migration is specified as an existing-target transformation, as in COPE (Listing 5.19), model elements and values that no longer conform to the target metamodel must be explicitly removed from the migrated model. Ecore2Ecore does not require explicit copying or unsetting code; instead, the relationship between original and evolved metamodel elements is captured in a mapping model specified by the metamodel developer. The mapping model can be derived automatically and customised by the metamodel developer. To explore the appropriateness for model migration of an alternative to new- and existing-target transformations, the following requirement was derived:

*The migration language must **automatically** copy every model element that conforms to the evolved metamodel from original to migrated model, and must automatically not copy any model element that does not conform to the evolved metamodel from original to migrated model.*

Model Representation Requirements

With Ecore2Ecore, migration is achieved by manipulating XMI. Consequently, the metamodel developer must be familiar with XMI and must perform tasks such as dereferencing URI fragments (Listing 5.18) and type conversion. Transformation languages abstract away from the underlying storage representation of models (such as XMI) by using a modelling framework to load, store and access models. Decoupling a transformation language from the model representation facilitates interoperability with more than one modelling technology, as demonstrated by the languages of the Epsilon platform [Kolovos 2009b]. 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. The editor can manipulate only metamodels defined with EMF. Similarly, the mapping tool used in the Ecore2Ecore approach can be used only with metamodels defined with EMF. Although

EMF is arguably very widely-used, other modelling frameworks exist. Adapting to interoperate with new systems is recognised as a common reason for software evolution [Sjøberg 1993], and migration between modelling frameworks is as a possible use case for a model migration language. To better support integration with modelling frameworks other than EMF, the following requirement was derived:

The migration language must be loosely coupled with modelling frameworks and must not assume that models and metamodels will be represented in EMF.

5.4 Epsilon Flock: A Model Migration Language

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

5.4.1 Design and Implementation

Flock has been designed to be a rule-based transformation language that mixes declarative and imperative parts. Consequently, Flock should be familiar to developers who have used hybrid-style M2M transformation languages, such as ATL and ETL [Kolovos *et al.* 2008b]. Flock has a compact syntax. 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 below.

Like Epsilon HUTN (Section 5.2.2), Flock is built atop Epsilon. In particular, Flock uses EMC to provide interoperability with several modelling frameworks, and EOL for specifying the imperative part of user-defined migration rules.

Abstract Syntax

As illustrated by Figure 5.19, Flock migration strategies are organised into modules (`FlockModule`). Flock modules inherit from EOL modules (`EolModule`) and hence provide language constructs for specifying user-defined operations and for re-using modules. Flock modules comprise any number of rules (`Rule`). Each rule has an original metamodel type (`originalType`) and can optionally specify a `guard`, which is either an EOL statement or a block of EOL statements. `MigrateRules` must specify an evolved metamodel type (`evolvedType`) and/or a `body` comprising a block of EOL statements.

```
1 migrate <originalType> (to <evolvedType>)?
2 (when (:<eolExpression>)|({<eolStatement>+}))? {
3   <eolStatement>*
4 }
5
6 delete <originalType>
7 (when (:<eolExpression>)|({<eolStatement>+}))?
```

Listing 5.21: Concrete syntax of migrate and delete rules.

Concrete Syntax

Listing 5.20 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` keywords. Migrate rules may also specify an evolved metamodel type using the `to` keyword and a `body` as a (possibly empty) sequence of EOL statements.

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

Execution Semantics

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

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

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

The rule selection phase has the following behaviour:

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

– Add the pair <r,e> to the set of selected rules.

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

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

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

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

Conservative Copy

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

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

- For each metafeature, f for which e has specified a value:
  - Find a metafeature, f′, of e′ with the same name as f.
    - If no equivalent metafeature can be found, do nothing.

- Otherwise, copy the original value (`e.f`) to produce a migrated value (`e'.f'`) if and only if the migrated value conforms to `f'`.

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

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

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 has been extended to include a conformance checking service; each EMC driver provides conformance checking semantics specific to its modelling framework. Specifically, EMC define Java interfaces for specifying the way in which model values are written to a model, and an additional, conformance-checking Java method has been added to the interface. When a Flock module is executed, conformance checking responsibilities are delegated to EMC drivers by calling the new method.

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

Development and User Tools

As discussed in Section 4.2, models and metamodels are typically kept separate. Flock migration strategies can be distributed by the metamodel developer in two ways. An extension point defined by Flock provides a generic user interface for migration strategy execution. Alternatively, metamodel developers can integrate model migration with other tools by accessing `FlockModule` programmatically. The latter approach facilities interoperability with, for example, model and source code management systems, and was used to provide a workflow architecture for the Epsilon languages in [Kolovos 2009b].

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

Listing 5.22: Redefining equivalences for the Component model migration.

5.4.2 Examples of Flock Migration

Flock is now demonstrated using three examples of model migration. The first example demonstrates the way in which a Flock module is executed and illustrates the semantics of conservative copy. The second describes the way in which the migration of the Petri net co-evolution example (Section 5.3.1) can be specified with Flock, and is included for direct comparison with the other languages discussed in Section 5.3. The final, larger example demonstrates all of the features of Flock, and is based on changes made to UML class diagrams between versions 1.5 and 2.0 of the UML specification.

Process-Oriented Migration in Flock

The first example considers the evolution of a process-oriented metamodel, introduced in 4.1.3 and described in . The process-oriented metamodel was developed to explore the feasibility of a graphical model editor for representing programs written in process-oriented programming languages, such as occam-π [Welch & Barnes 2005].

The original metamodel, shown in Figure 5.20(a), has been evolved to distinguish between `ConnectionPoints` that are a reader for a `Channel` and `ConnectionPoints` that are a writer for a `Channel` by making `ConnectionPoint` abstract and introducing two subtypes, `ReadingConnectionPoint` and `WritingConnectionPoint`, as shown in Figure 5.20(b).

The model shown in Figure 5.21 conforms to the original metamodel in Figure 5.20(a) and is to be migrated. The model comprises three `Processes` named *delta*, *prefix* and *minus*; three `Channels` named *a*, *b* and *c*; and six `ConnectionPoints` named *a?*, *a!*, *b?*, *b!*, *c?* and *c!*.

For the migration strategy shown in Listing 5.21, the Flock module will perform the following steps. Firstly, the rule selection phase produces a set of pairs `<r,e>`. For each `ConnectionPoint`, the guard part of the user-defined rules control which rule will be selected. `ConnectionPoints a!`, `b!` and `c!` have outgoing `Channels` (`a`, `b` and `c` respectively) and hence the migration rule on line 1 is selected. Similarly, the `ConnectionPoints a?`, `b?` and `c?` have incoming `Channels` (`a`, `b` and `c` respectively) and hence the migration rule on line 2 is selected. There is no `ConnectionPoint` with both an outgoing and an incoming `Channel`, but if there were, the first applicable rule (i.e. the rule on line 1) would be selected. For the other model

elements (the Processes and Channels) no user-defined rules are applicable, and so default rules are used instead. A default rule has an empty body and identical original and evolved types. In other words, a default rule for the Process type is equivalent to the user-defined rule: `migrate Process to Process {}`

Secondly, the equivalence establishment phase creates an element, e′, in the migrated model for each pair <r,e>. For each ConnectionPoint, the evolved type of the selected rule (r) controls the type of e′. The rule on line 1 of Listing 5.21 was selected for the ConnectionPoints a!, b! and c! and hence an equivalent element of type ReadingConnectionPoint is created for a!, b! and c!. Similarly, an equivalent element of type WritingConnectionPoint is created for a?, b? and c?. For the other model elements (the Processes and Channels) a default rule was selected, and hence the equivalent model element has the same type as the original model element.

Finally, the rule execution phase performs a conservative copy for each original and equivalent model element in the set of triples <r,e,e′> produced by the equivalent establishment phase. The metamodel evolution shown in Figure 5.20 has not affected the Process type, and hence for each Process in the original model, conservative copy will create a Process in the migrated model and copy the values of all features. For each Channel in the original model, conservative copy will create an equivalent Channel in the migrated model and copy the value of the name feature from original to migrated model element. However, the values of the reader and writer features will not be copied by conservative copy because the type of these features has changed (from ConnectionPoint to ReadingConnectionPoint and WritingConnectionPoint, respectively). The values of the reader and writer features in the original model will not conform to the reader and writer features in the evolved metamodel. Finally, the values of the name, incoming and outgoing features of the ConnectionPoint class have not evolved, and hence are copied directly from original to equivalent model elements.

The rule execution phase also executes the body of each rule, r, for every triple in the set <r,e,e′>. The user-defined rules in Listing 5.21 have no body, and hence no further execution is performed in this case.

Petri Nets Migration in Flock

The Petri net metamodel evolution demonstrates the core functionality of Flock. In Listing 5.22, Nets and Places are migrated automatically. Unlike the ATL migration strategy (Listing 5.17), no explicit copying rules are required. Compared to the COPE migration strategy (Listing 5.19), the Flock migration strategy does not need to 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 5.23: Petri nets model migration in Flock

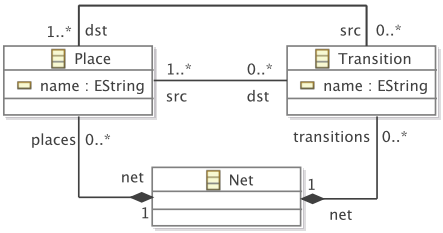
UML Class Diagram Migration in Flock

Figure 5.22 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 5.22(a), association ends and attributes are specified separately. In Figure 5.22(b), the `Property` class is used instead. The Flock migration strategy (Listing 5.23) for Figure 5.22 is now discussed.

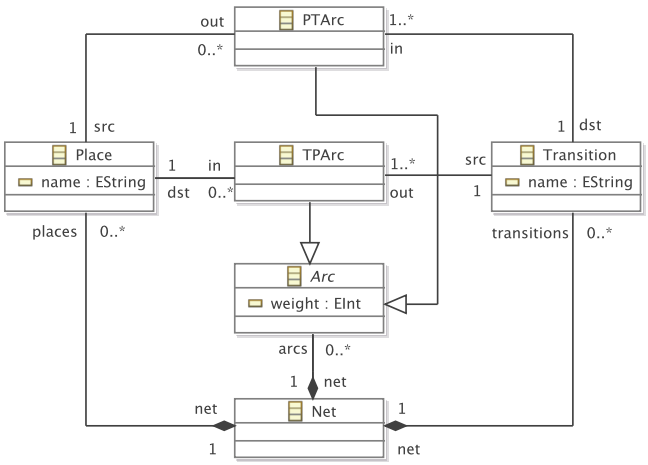
```
migrate Association {
  migrated.memberEnds := original.connections.equivalent();
}

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;
  }
}

```

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

Listing 5.24: UML model migration in Flock



(a) Original metamodel.



(b) Evolved metamodel.

Figure 5.19: Exemplar metamodel evolution. Taken from [Rose *et al.* 2010f].

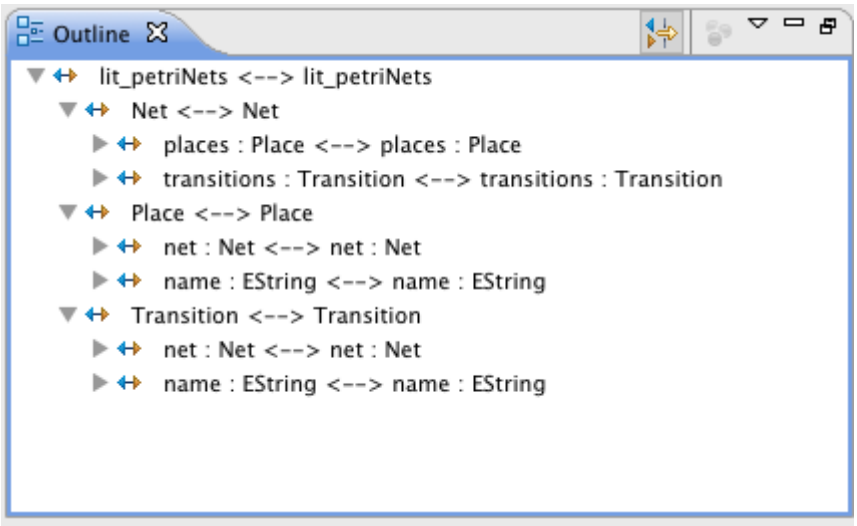


Figure 5.20: Mappings between the original and evolved Petri nets metamodels, constructed with the tool described in [Hussey & Paternostro 2006]

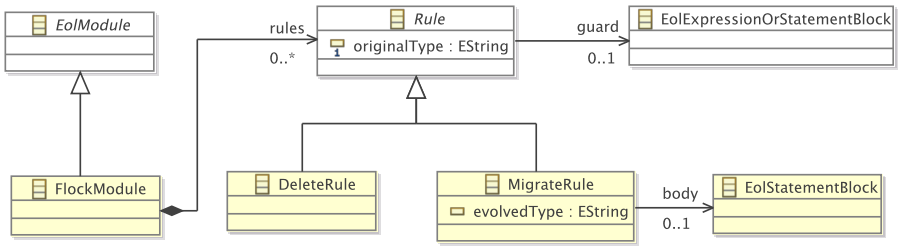
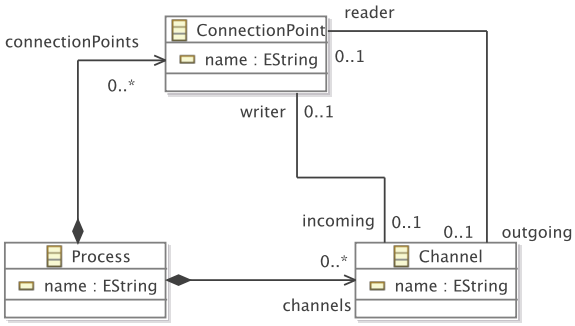
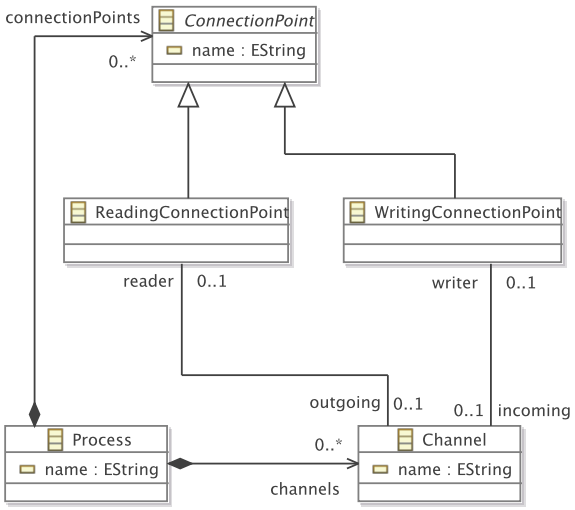


Figure 5.21: The abstract syntax of Flock.



(a) Original metamodel.



(b) Evolved metamodel.

Figure 5.22: Evolution of the Process-Oriented metamodel (Appendix A)

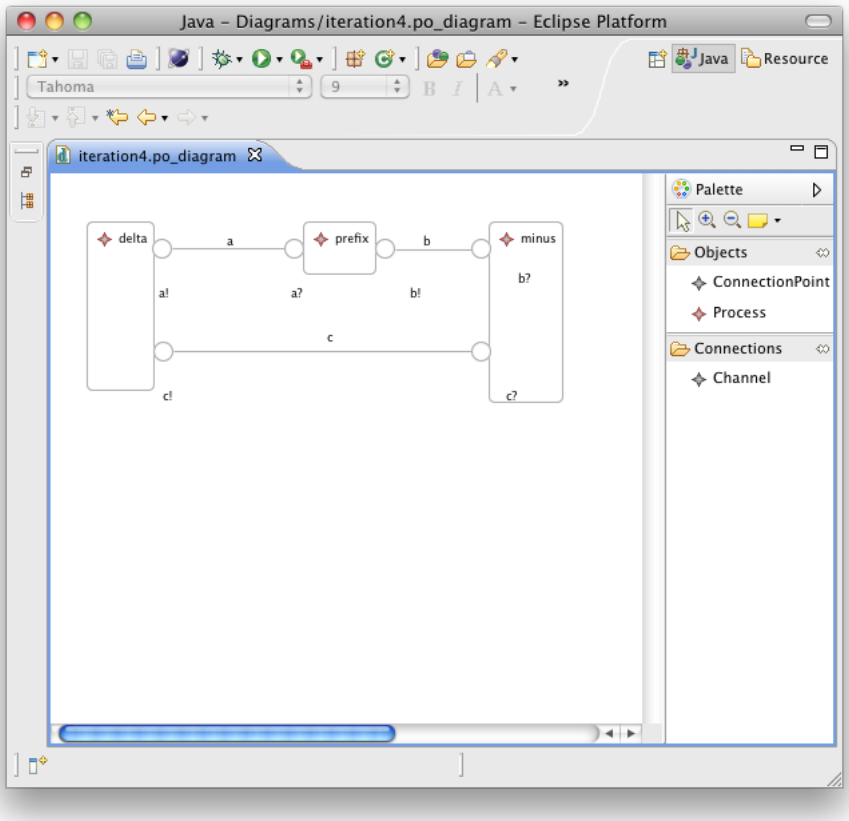
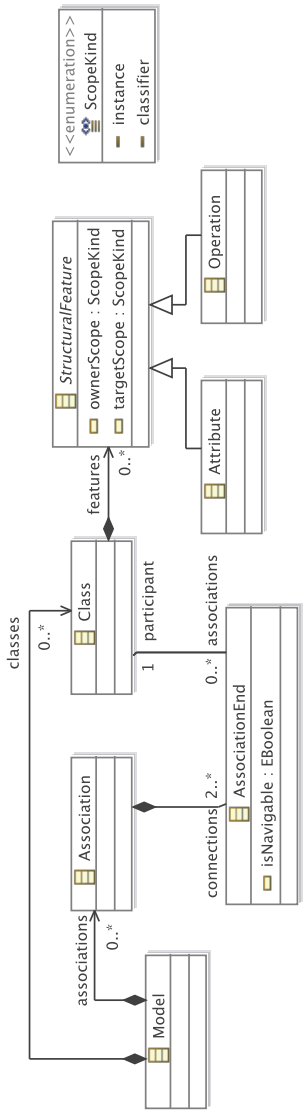
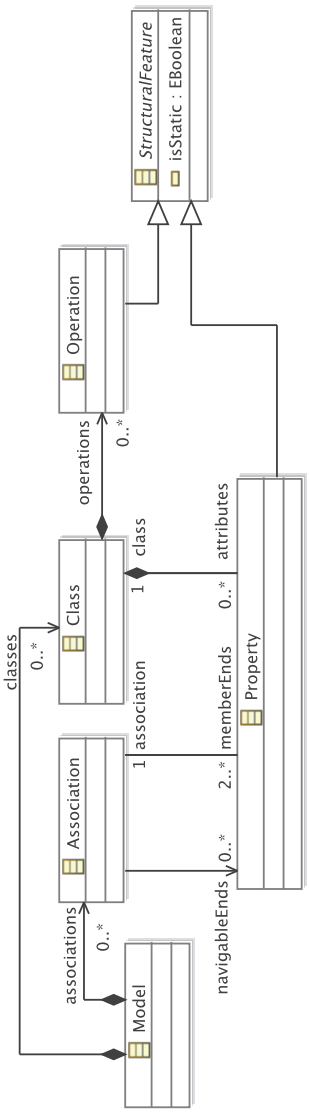


Figure 5.23: Process-Oriented model prior to migration



(a) Original, UML 1.5 metamodel.



(b) Evolved, UML 2.0 metamodel.

Figure 5.24: UML metamodel evolution

Firstly, `Attributes` and `AssociationEnds` are migrated to be `Properties` (lines 14 to 18, and 25 to 29). In particular, 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 26-28).

In UML 2.0, `StructuralFeature#ownerScope` has been replaced by `#isStatic` (lines 15-17 and 20-22). The UML 2.0 specification states that the UML 1.5 values `ScopeKind#classifier` and `#instance` should be migrated to true and to false, respectively.

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 7 and 8 partition features on the original `Class` into `Operations` and `Propertys`. `Class#associations` has been removed in UML 2.0, and `AssociationEnds` are instead stored in `Class#attributes` (line 9).

Summary

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

All of the approaches considered in Section 5.3 support EMF. Both Flock and ATL support further modelling technologies, such as MDR and XML. However, ATL does not automatically copy model elements that have not been affected by metamodel changes. Therefore, migration between models of different technologies with ATL requires extra statements in the migration strategy to ensure that the conformance constraints of the target technology are satisfied. Because it delegates conformance checking to an EMC driver, Flock requires no such checks.

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

5.5 Chapter Summary

Three structures for identifying and managing co-evolution have been designed and implemented to approach the thesis requirements outlined in Chapter 4. The way in which modelling frameworks implicitly enforce conformance makes

managing non-conformant models challenging, and the proposed metamodel-independent syntax (Section 5.1) extends modelling frameworks to facilitate the management of non-conformant models. The proposed textual modelling notation, Epsilon HUTN (Section 5.2), provides a human-usable notation as an alternative to XMI for performing user-driven co-evolution. Finally, Epsilon Flock (Section 5.4) contributes a domain-specific language for describing model migration.

The metamodel-independent syntax is a modelling framework extension that makes explicit the conformance relationship between models and metamodels. By binding models not to their metamodel but to a generic metamodel, the metamodel-independent syntax allows non-conformant models to be managed with modelling tools and model management operations. Furthermore, conformance checking is provided as a service, which can be scheduled at any time, and not just when models are loaded. The metamodel-independent syntax has been integrated with Concordance [Rose *et al.* 2010c] to provide a metamodel installation process that automatically reports conformance problems, and underpins the implementation of the second structure described in this chapter, a textual modelling notation.

For performing user-driven co-evolution, the textual modelling notation described in Section 5.2 provides an alternative to XMI. Unlike XMI, the notation introduced in this chapter implements the OMG standard for Human-Usable Textual Notation (HUTN) [OMG 2004] and is optimised for human usability. Epsilon HUTN, introduced here, is presently the sole reference implementation of HUTN. Constructing Epsilon HUTN atop the metamodel-independent syntax allows Epsilon HUTN to provide incremental and background conformance checking, and an XMI-to-HUTN transformation for loading non-conformant models. Section 6.1 explores the benefits and drawbacks of using the metamodel-independent syntax and Epsilon HUTN together to perform user-driven co-evolution.

The domain-specific language described in Section 5.4, Epsilon Flock, combines several concepts from existing model-to-model transformation languages to form a language tailored to model migration. In particular, Flock contributes a novel mechanism for relating source and target model elements termed conservative copy, which is a hybrid of new- and existing-target styles

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

Table 5.2: Properties of model migration approaches

of model-to-model transformation. Flock is built atop Epsilon and hence interoperates transparently with several modelling technologies via the Epsilon Model Connectivity layer.

The metamodel-independent syntax, Epsilon HUTN, Epsilon Flock and Concordance have been released as part of Epsilon in the Eclipse GMT[4] project, which is the research incubator of arguably the most widely used MDE modelling framework, EMF. By re-using parts of Epsilon, the structures were implemented more rapidly than would have been possible when developing the structures independently. In particular, re-using the Epsilon Model Connectivity layer facilitated interoperability of Flock with several MDE modelling frameworks, which was exploited to manage a practical case of model migration in Section 6.4.

[4] http://www.eclipse.org/gmt

Bibliography

[Ackoff 1962] R.L. Ackoff. *Scientific Method: Optimizing Applied Research Decisions*. John Wiley and Sons, New York, 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. Workshop on Traceability, co-located with the European Conference on Model-Driven Architecture (ECMDA)*, pages 8–14, 2005.

[Alexander *et al.* 1977] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series)*. Oxford University Press, New York, 1977.

[Álvarez *et al.* 2001] J. Álvarez, A. Evans, and P. Sammut. MML and the metamodel architecture. In *Proc. Workshop on Transformation in UML, co-located with the European Joint Conferences on Theory and Practice of Software (ETAPS)*, 2001.

[Amyot *et al.* 2006] D. Amyot, H. Farah, and J.-F. Roy. Evaluation of development tools for domain-specific modeling languages. In R. Gotzhein and R. Reed, editors, *System Analysis and Modeling: Language Profiles*, volume 4320 of *Lecture Notes in Computer Science*, pages 183–197. Springer, Heidelberg, 2006.

[Apostel 1960] L. Apostel. Towards the formal study of models in the non-formal sciences. *Synthese*, 12(2):125–161, 1960.

[Arendt *et al.* 2009] T. Arendt, F. Mantz, L. Schneider, and G. Taentzer. Model refactoring in Eclipse by LTK, EWL, and EMF Refactor: A case study. In *Proc. Joint Model-Driven Software Evolution and Model Co-evolution and Consistency Management (MoDSE-MCCM) Workshop, co-located with the International Conference on Model-Driven Engineering Languages and Systems (MoDELS)*, 2009.

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

[Balazinska *et al.* 2000] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proc. Working Conference on Reverse Engineering (WCRE)*, pages 98–107. IEEE Computer Society, 2000.

[Banerjee *et al.* 1987] J. Banerjee, W. Kim, H. Kim, and H.F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In U. Dayal and I.L. Traiger, editors, *Proc. Special Interest Group on Management of Data (SIGMOD) Conference*, pages 311–322. ACM Press, 1987.

[Barry 2006] B. Barry. A view of 20th and 21st century software engineering. In L.J. Osterweil, H.D. Rombach, and M.L. Soffa, editors, *Proc. International conference on Software Engineering (ICSE)*, pages 12–29. ACM, 2006.

[Beattie *et al.* 2007] B.R. Beattie, C.R. Taylor, and M.J. Watts. *The Economics of Production.* Krieger Publishing Company, 2nd edition, 2007.

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

[Bézivin & Gerbé 2001] J. Bézivin and O. Gerbé. Towards a precise definition of the OMG/MDA framework. In *Proc. International Conference on Automated Software Engineering (ASE)*, pages 273–280. IEEE Computer Society, 2001.

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

[Biermann *et al.* 2006] E. Biermann, K. Ehrig, C. Köhler, G. Kuhns, G. Taentzer, and E. Weiss. EMF model refactoring based on graph transformation concepts. *Electronic Communications of the European Association for the Study of Science and Technology [online]*, 3, 2006. Available at `http://eceasst.cs.tu-berlin.de/index.php/eceasst/article/view/34` [Accessed 2 November 2010].

[Bloch 2005] J. 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. Available at: `http://lcsd05.cs.tamu.edu/slides/keynote.pdf` [Accessed 2 November 2010], 2005.

[Bohner 2002] S.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] J. Bosch. Design patterns as language constructs. *Journal of Object Oriented Programming*, 11(2):18–32, 1998.

[Briand *et al.* 2003] L.C. Briand, Y. 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 Jr. 1987] F.P. Brooks Jr. No silver bullet - essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.

[Brooks Jr. 1995] F.P. Brooks Jr. *The mythical man-month.* Addison-Wesley Longman Publishing Co., Inc., Boston, Massachusetts, 25th anniversary edition, 1995.

[Brown *et al.* 1998] W.J. Brown, R.C. Malveau, H.W. McCormick III, and T.J. Mowbray. *Anti Patterns.* Wiley, New York, 1998.

[Cervelle *et al.* 2006] J. Cervelle, R. Forax, and G. Roussel. Tatoo: an innovative parser generator. In Ralf Gitzel, Markus Aleksy, and Martin Schader, editors, *Proc. International Symposium on Principles and Practice of Programming in Java (PPPJ)*, volume 178 of *ACM International Conference Proceeding Series*, pages 13–20. ACM, 2006.

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

[Cicchetti *et al.* 2008] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Automating co-evolution in model-driven engineering. In *Proc. International IEEE Enterprise Distributed Object Computing Conference (EDOC)*, pages 222–231. IEEE Computer Society, 2008.

[Cicchetti 2008] A. 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] T. Clark, P. Sammut, and J. Willians. *Superlanguages: Developing Languages and Applications with XMF [online].* Ceteva, Sheffield, 2008. Available at: `http://itcentre.tvu.ac.uk/~clark/Papers/Superlanguages.pdf` [Accessed 02 November 2010].

[Cleland-Huang *et al.* 2003] J. Cleland-Huang, C.K. Chang, and M. 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. Workshop on Traceability, co-located with the European Conference on Model-Driven Architecture (ECMDA)*, pages 17–26, 2007.

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

[Deursen *et al.* 2000] A. van Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An annotated bibliography. *Special Interest Group on Programming Languages (SIGPLAN) Notices*, 35(6):26–36, 2000.

[Deursen *et al.* 2007] A. van Deursen, E. Visser, and J. Warmer. Model-driven software evolution: A research agenda. In *Proc. Workshop on Model-Driven Software Evolution, co-located with the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 41–49, 2007.

[Dig & Johnson 2006a] D. Dig and R. Johnson. Automated upgrading of component-based applications. In P.L. Tarr and W.R. Cook, editors, *Companion to the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 675–676, 2006.

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

[Dig *et al.* 2006] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In D. Thomas, editor, *Proc. European Conference on Object-Oriented Programming (ECOOP)*, volume 4067 of *Lecture Notes in Computer Science*, pages 404–428. Springer, 2006.

[Dig *et al.* 2007] D. Dig, K. Manzoor, R. Johnson, and T.N. Nguyen. Refactoring-aware configuration management for object-oriented programs. In *Proc. International Conference on Software Engineering (ICSE)*, pages 427–436. IEEE Computer Society, 2007.

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

[Drivalos *et al.* 2008] N. Drivalos, R.F. Paige, K.J. Fernandes, and D.S. Kolovos. Towards rigorously defined model-to-model traceability. In *Proc. Workshop on Traceability, co-located with the European Conference on Model Driven Architecture (ECMDA)*, 2008.

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

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

[Elmasri & Navathe 2006] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley Longman, Boston, Massachusetts, 5th edition, 2006.

[Erlikh 2000] L. 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, Massachusetts, 2004.

[Feathers 2004] M.C. Feathers. *Working Effectively with Legacy Code*. Prentice Hall, Upper Saddle River, New Jersey, 2004.

[Ferrandina *et al.* 1995] F. Ferrandina, T. Meyer, R. Zicari, G. Ferran, and J. Madec. Schema and database evolution in the O2 object database system. In U. Dayal, P.M.D. Gray, and S. Nishio, editors, *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 170–181. Morgan Kaufmann, 1995.

[Fowler 1999] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley, Upper Saddle River, New Jersey, 1999.

[Fowler 2002] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, Boston, Massachusetts, 2002.

[Fowler 2010] M. Fowler. *Domain Specific Languages*. Addison-Wesley Professional, Boston, Massachusetts, 2010.

[Frankel 2002] D. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons Inc., New York, 2002.

[Fritzsche *et al.* 2008] M. Fritzsche, J. Johannes, S. Zschaler, A. Zherebtsov, and A. Terekhov. Application of tracing techniques in model-driven performance engineering. In *Proc. Traceability Workshop, co-located with the European Conference on Model-Driven Architecture - Foundations and Applications (ECMDA-FA)*, pages 111–120, 2008.

[Fuhrer *et al.* 2007] R.M. Fuhrer, A. Kiezun, and M. Keller. Refactoring in the Eclipse JDT: Past, present, and future. In *Proc. Workshop on Refactoring Tools (WRT), co-located with European Conference on Object-Oriented Programming (ECOOP)*, 2007.

[Gamma *et al.* 1995] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, Reading, Massachusetts, 1995.

[Garcés *et al.* 2009] K. Garcés, F. Jouault, P. Cointe, and J. Bézivin. Managing model adaptation by precise detection of metamodel changes. In R.F. Paige, A. Hartman, and A. Rensink, editors, *Proc. European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA)*, volume 5562 of *Lecture Notes in Computer Science*, pages 34–49. Springer, 2009.

[Geiß & Kroll 2007] R. Geiß and M. Kroll. GrGen.NET: A fast, expressive, and general purpose graph rewrite tool. In A. Schürr, M. Nagl, and A. Zündorf, editors, *Proc. International Symposium on Applications of Graph Transformations with Industrial Relevance (AGTIVE), Revised Selected and Invited Papers*, volume 5088 of *Lecture Notes in Computer Science*, pages 568–569. Springer, 2007.

[Gosling *et al.* 2005] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java^{TM} Language Specification*. Addison-Wesley, Boston, Massachusetts, 2005.

[Graham 1993] P. Graham. *On Lisp: Advanced Techniques for Common Lisp*. Prentice-Hall, Upper Saddle River, New Jersey, 1993.

[Greenfield *et al.* 2004] J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley & Sons Inc., New York, 2004.

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

[Gruschko *et al.* 2007] B. Gruschko, D.S. Kolovos, and R.F. Paige. Towards synchronizing models with evolving metamodels. In *Proc. Workshop on Model-Driven Software Evolution, co-located with the European Conference on Software Maintenance and Reengineering (CSMR)*, 2007.

[Guerrini *et al.* 2005] G. Guerrini, M. Mesiti, and D. Rossi. Impact of XML schema evolution on valid documents. In A. Bonifati and D. Lee, editors, *Proc. International Workshop on Web Information and Data Management (WIDM)*, pages 39–44. ACM, 2005.

[Halstead 1977] M.H. Halstead. *Elements of Software Science*. Elsevier Science Inc., New York, 1977.

[Hearnden *et al.* 2006] D. Hearnden, M. Lawley, and K. Raymond. Incremental model transformation for the evolution of model-driven systems. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *Proc. International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 4199 of *Lecture Notes in Computer Science*, pages 321–335. Springer, 2006.

[Heidenreich *et al.* 2009] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende. Derivation and refinement of textual syntax for models. In R.F. Paige, A. Hartman, and A. Rensink, editors, *Proc. European Conference on Model-Driven Architecture - Foundations and Applications (ECMDA-FA)*, volume 5562 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 2009.

[Herrmannsdoerfer *et al.* 2008a] M. Herrmannsdoerfer, S. Benz, and E. Juergens. Automatability of coupled evolution of metamodels and models in practice. In K. Czarnecki, I. Ober, J. Bruel, A. Uhl, and M. Völter, editors, *Proc. International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 5301 of *Lecture Notes in Computer Science*, pages 645–659. Springer, 2008.

[Herrmannsdoerfer *et al.* 2008b] M. Herrmannsdoerfer, S. Benz, and E. Juergens. COPE: A language for the coupled evolution of metamodels and models. In *Proc. International Workshop on Model Co-Evolution and Consistency Management (MCCM), co-located with the International Conference on Model-Driven Engineering Languages and Systems (MoDELS)*, 2008.

[Herrmannsdoerfer *et al.* 2009a] M. Herrmannsdoerfer, S. Benz, and E. Juergens. COPE - automating coupled evolution of metamodels and models. In S. Drossopoulou, editor, *Proc. European Conference on Object-Oriented Programming (ECOOP)*, volume 5653 of *Lecture Notes in Computer Science*, pages 52–76. Springer, 2009.

[Herrmannsdoerfer *et al.* 2009b] M. Herrmannsdoerfer, D. Ratiu, and G. Wachsmuth. Language evolution in practice. In M. van den Brand, D. Gasevic, and J. Gray, editors, *Proc. International Conference on Software Language Engineering (SLE), Revised Selected Papers*, volume 5696 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2009.

[Hussey & Paternostro 2006] K. Hussey and M. Paternostro. Advanced features of EMF. Tutorial at EclipseCon 2006, California, USA. [Accessed 02 November 2010] Available at: `http://www.eclipsecon.org/2006/Sub.do?id=171`, 2006.

[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, New York, 1995.

[Jouault & Kurtev 2005] F. Jouault and I. Kurtev. Transforming models with ATL. In J-M. Bruel, editor, *Proc. Satellite Events at the International Conference on Model Driven Engineering Languages and Systems (MoDELS), International Workshops, Doctoral Symposium, Educators Symposium, Revised Selected Papers*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer, 2005.

[Jouault 2005] F. Jouault. Loosely coupled traceability for ATL. In *Proc. Workshop on Traceability, co-located with the European Conference on Model-Driven Architecture (ECMDA)*, 2005.

[Jurack & Mantz 2010] S. Jurack and F. Mantz. Towards metamodel evolution of EMF models with Henshin. In *Proc. Models and Evolution Workshop, co-located with the International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, 2010.

[Kalnins *et al.* 2005] A. Kalnins, J. Barzdins, and E. Celms. Model transformation language MOLA. In U. Aßmann, M. Aksit, and A. Rensink, editors, *Proc. Model Driven Architecture, European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004, Revised Selected Papers*, volume 3599 of *Lecture Notes in Computer Science*, pages 62–76. Springer, 2005.

[Kataoka *et al.* 2001] Y. Kataoka, M.D. Ernst, W.G. Griswold, and D. Notkin. Automated support for program refactoring using invariants. In *Proc. International Conference on Software Maintenance (ICSM)*, pages 736–743. IEEE Computer Society, 2001.

[Kelly & Tolvanen 2008] S. Kelly and J.P. Tolvanen. *Domain-Specific Modelling.* Wiley and IEEE Computer Society, 2008.

[Kelly 1999] T.P. Kelly. *Arguing Safety – A Systematic Approach to Safety Case Management.* PhD thesis, University of York, United Kingdom, 1999.

[Kerievsky 2004] J. Kerievsky. *Refactoring to Patterns.* Pearson Higher Education, 2004.

[Kleppe *et al.* 2003] A.G. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co. Inc., Boston, Massachusetts, 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(3):331–380, 2003.

[Kolovos *et al.* 2006a] D.S. Kolovos, R.F. Paige, and F.A. Polack. The Epsilon Object Language (EOL). In A. Rensink and J. Warmer, editors, *Proc. European Conference on Model-Driven Architecture - Foundations and Applications (ECMDA-FA)*, volume 4066 of *Lecture Notes in Computer Science*, pages 128–142. Springer, 2006.

[Kolovos *et al.* 2006b] D.S. Kolovos, R.F. Paige, and F.A.C. Polack. Merging models with the Epsilon Merging Language (EML). In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *Proc. International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 4199 of *Lecture Notes in Computer Science*, pages 215–229. Springer, 2006.

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

[Kolovos *et al.* 2008a] D.S. Kolovos, R.F. Paige, and F.A.C. Polack. The grand challenge of scalability for model driven engineering. In M.R. Chaudron, editor, *Models in Software Engineering: Workshops and Symposia at MoDELS 2008, Reports and Revised Selected Papers*, volume 5421 of *Lecture Notes in Computer Science*, pages 48–53. Springer-Verlag, 2008.

[Kolovos *et al.* 2008b] D.S. Kolovos, R.F. Paige, and F.A.C Polack. The Epsilon Transformation Language. In A. Vallecillo, J. Gray, and A. Pierantonio, editors, *Proc. International Conference on the Theory and Practice of Model Transformations (ICMT)*, volume 5063 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2008.

[Kolovos *et al.* 2009] D.S. Kolovos, R.F. Paige, and F.A.C. Polack. On the evolution of OCL for capturing structural constraints in modelling languages. In J. Abrial and U. Glässer, editors, *Rigorous Methods for Software Construction and Analysis*, volume 5115 of *Lecture Notes in Computer Science*, pages 204–218. Springer, 2009.

[Kolovos *et al.* 2010] D.S. Kolovos, L.M. Rose, S. bin Abid, R.F. Paige, F.A.C. Polack, and G. Botterweck. Taming EMF and GMF using model transformation. In D.C. Petriu, N. Rouquette, and Ø. Haugen, editors, *Proc. International Conference on Model Driven Engineering Languages*

and System (MoDELS), Part I, volume 6394 of *Lecture Notes in Computer Science*, pages 211–225. Springer, 2010.

[Kolovos 2009a] D.S. Kolovos. Establishing correspondences between models with the Epsilon Comparison Language. In R.F. Paige, A. Hartman, and A. Rensink, editors, *Proc. European Conference on Model-Driven Architecture - Foundations and Applications (ECMDA-FA)*, volume 5562 of *Lecture Notes in Computer Science*, pages 146–157. Springer, 2009.

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

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

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

[Lago *et al.* 2009] P. Lago, H. Muccini, and H. 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 J.N. Oliveira and P. Zave, editors, *Proc. Formal Methods for Increasing Software Productivity (FME), International Symposium of Formal Methods Europe*, volume 2021 of *Lecture Notes in Computer Science*, pages 550–570. Springer, 2001.

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

[Lara & Guerra 2010] J. de Lara and E. Guerra. Generic meta-modelling with concepts, templates and mixin layers. In D.C. Petriu, N. Rouquette, and Ø. Haugen, editors, *Proc. International Conference on Model Driven Engineering Languages and System (MoDELS), Part I*, volume 6394 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 2010.

[Lehman 1969] M.M. Lehman. The programming process. Technical report, IBM Research Report RC 2722, 1969.

[Lerner 2000] B.S. 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. IEEE Computer Society, 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] J. McCarthy. History of Lisp. *History of Programming Languages*, 1:217–223, 1978.

[McNeile 2003] A. McNeile. MDA: The vision with the hole? [online]. *Metamaxim Ltd*, 2003. [Accessed 02 November 2010] Available at: `http://www.metamaxim.com/download/documents/MDAv1.pdf`.

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

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

[Méndez *et al.* 2010] D. Méndez, A. Etien, A. Muller, and R. Casallas. Towards transformation migration after metamodel evolution. In *Proc. Models and Evolution Workshop, co-located with the International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, 2010.

[Mens & Demeyer 2007] T. Mens and S. Demeyer. *Software Evolution*. Springer-Verlag, Berlin, 2007.

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

[Mens *et al.* 2007] T. Mens, G. Taentzer, and D. Müller. Challenges in model refactoring. In *Proc. Workshop on Object-Oriented Reengineering, co-located with the European Conference on Object-Oriented Programming (ECOOP)*, 2007.

[Merriam-Webster 2010] Merriam-Webster. Definition of Nuclear Family [online]. [Accessed 02 November 2010] Available at: `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] N. Moha, V. Mahé, O. Barais, and J.M. Jézéquel. Generic model refactorings. In A. Schürr and B. Selic, editors, *Proc. International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 5795 of *Lecture Notes in Computer Science*, pages 628–643. Springer, 2009.

[Muller & Hassenforder 2005] P. Muller and M. Hassenforder. HUTN as a Bridge between ModelWare and GrammarWare. In *Proc. Workshop in Software Modelling Engineering, co-located with the International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, 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] T.N. Nguyen, C. Thao, and E.V. Munson. On product versioning for hypertexts. In *Proc. International Workshop on Software Configuration Management (SCM)*, pages 113–132. ACM, 2005.

[Nickel *et al.* 2000] U. Nickel, J. Niere, and A. Zündorf. The FUJABA environment. In *Proc. International Conference on Software Engineering (ICSE)*, pages 742–745. ACM, 2000.

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

[Oldevik *et al.* 2005] J. Oldevik, T. Neple, R. Grønmo, J.Ø. Aagedal, and A. Berre. Toward standardised model to text transformations. In A. Hartman and D. Kreische, editors, *Proc. European Conference on Model-Driven Architecture - Foundations and Applications (ECMDA-FA)*, volume 3748 of *Lecture Notes in Computer Science*, pages 239–253. Springer, 2005.

[Olsen & Oldevik 2007] G.K. Olsen and J. Oldevik. Scenarios of traceability in model to text transformations. In D.H. Akehurst, R. Vogel, and R.F. Paige, editors, *Proc. European Conference on Model-Driven Architecture - Foundations and Applications (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 02 November 2010] Available at: `http://www.omg.org/spec/UML/1.4/`, 2001.

[OMG 2004] OMG. Human-Usable Textual Notation 1.0 Specification [online]. [Accessed 02 November 2010] 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 02 November 2010] Available at: `http://www.omg.org/technology/documents/formal/ocl.htm`, 2006.

[OMG 2007a] OMG. Unified Modelling Language 2.1.2 Specification [online]. [Accessed 02 November 2010] Available at: `http://www.omg.org/spec/UML/2.1.2/`, 2007.

[OMG 2007b] OMG. Unified Modelling Language 2.2 Specification [online]. [Accessed 02 November 2010] Available at: `http://www.omg.org/spec/UML/2.2/`, 2007.

[OMG 2007c] OMG. XML Metadata Interchange 2.1.1 Specification [online]. [Accessed 02 November 2010] Available at: `http://www.omg.org/technology/documents/formal/xmi.htm`, 2007.

[OMG 2008a] OMG. Meta-Object Facility [online]. [Accessed 02 November 2010] Available at: `http://www.omg.org/mof`, 2008.

[OMG 2008b] OMG. Model Driven Architecture [online]. [Accessed 02 November 2010] Available at: `http://www.omg.org/mda/`, 2008.

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

[Paige *et al.* 2007] R.F. Paige, P.J. Brooke, and J.S. Ostroff. Metamodel-based model conformance and multiview consistency checking. *ACM Transactions on Software Engineering and Methodology*, 16(3):1–48, 2007.

[Paige *et al.* 2009] R.F. Paige, L.M. Rose, X. Ge, D.S. Kolovos, and P.J. Brooke. FPTC: Automated safety analysis for domain-specific languages. In M.R.V. Chaudron, editor, *Models in Software Engineering, Workshops and Symposia at MoDELS 2008, Reports and Revised Selected Papers*, volume 5421 of *Lecture Notes in Computer Science*, pages 229–242. Springer, 2009.

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

[Patrascoiu & Rodgers 2004] O. Patrascoiu and P. Rodgers. Embedding OCL expressions in YATL. In *Proc. OCL and Model-Driven Engineering Workshop, co-located with the International Conference on the Unified Modeling Language (UML)*, 2004.

[Pilgrim *et al.* 2008] J. von Pilgrim, B. Vanhooff, I. Schulz-Gerlach, and Y. Berbers. Constructing and visualizing transformation chains. In I. Schieferdecker and A. Hartman, editors, *Proc. European Conference on the Model Driven Architecture – Foundations and Applications*, volume 5095 of *Lecture Notes in Computer Science*, 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] I. Porres. Model refactorings as rule-based update transformations. In P. Stevens, J. Whittle, and G. Booch, editors, *Proc. International Conference on the Unified Modeling Language, Modeling Languages and Applications (UML)*, volume 2863 of *Lecture Notes in Computer Science*, 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] J.F. Ramil and M.M. Lehman. Cost estimation and evolvability monitoring for software evolution processes. In *Proc. Workshop on Empirical Studies of Software Maintenance (WESS), co-located with the International Conference on Software Maintenance*, 2000.

[Ráth *et al.* 2008] I. Ráth, G. Bergmann, A. Ökrös, and D. Varró. Live model transformations driven by incremental pattern matching. In A. Vallecillo, J. Gray, and A. Pierantonio, editors, *Proc. International Conference on the Theory and Practice of Model Transformations (ICMT)*, volume 5063 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2008.

[Rios *et al.* 2006] E. Rios, T. Bozheva, A. Bediaga, and N. Guilloreau. MDD maturity model: A roadmap for introducing Model-Driven Development. In A. Rensink and J. Warmer, editors, *Proc. European Conference on Model Driven Architecture Foundations and Applications (ECMDA-FA)*, volume 4066 of *Lecture Notes in Computer Science*, pages 78–89. Springer Berlin / Heidelberg, 2006.

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

[Rose *et al.* 2008a] L.M. Rose, R.F. Paige, D.S. Kolovos, and F.A.C. Polack. Constructing models with the Human-Usable Textual Notation. In K. Czarnecki, I. Ober, J. Bruel, A. Uhl, and M. Völter, editors, *Proc. International Conference on Model Driven Engineering Languages and*

Systems (MoDELS), volume 5301 of *Lecture Notes in Computer Science*, pages 249–263. Springer, 2008.

[Rose *et al.* 2008b] L.M. Rose, R.F. Paige, D.S. Kolovos, and F.A.C. Polack. The Epsilon Generation Language. In I. Schieferdecker and A. Hartman, editors, *Proc. European Conference on Model Driven Architecture – Foundations and Applications*, volume 5095 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2008.

[Rose *et al.* 2009a] L.M. Rose, D.S. Kolovos, R.F. Paige, and F.A.C. Polack. Enhanced automation for managing model and metamodel inconsistency. In *Proc. International Conference on Automated Software Engineering (ASE)*, pages 545–549. IEEE Computer Society, 2009.

[Rose *et al.* 2009b] L.M. Rose, R.F. Paige, D.S. Kolovos, and F.A.C. Polack. An analysis of approaches to model migration. In *Proc. Joint Model-Driven Software Evolution and Model Co-evolution and Consistency Management (MoDSE-MCCM) Workshop, co-located with the International Conference on Model-Driven Engineering Languages and Systems (MoDELS)*, 2009.

[Rose *et al.* 2010a] L.M. Rose, A. Etien, D. Méndez, D.S. Kolovos, R.F. Paige, and F.A.C. Polack. Comparing model-metamodel and transformation-metamodel co-evolution. In *Proc. Models and Evolution Workshop, co-located with the International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, 2010.

[Rose *et al.* 2010b] L.M. Rose, M. Herrmannsdoerfer, J.R. Williams, D.S. Kolovos, K. Garcés, R.F. Paige, and F.A.C. Polack. A comparison of model migration tools. In D.C. Petriu, N. Rouquette, and Ø Haugen, editors, *Proc. International Conference on Model Driven Engineering Languages and Systems (MoDELS), Part I*, volume 6394 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2010.

[Rose *et al.* 2010c] L.M. Rose, D.S. Kolovos, N. Drivalos, J.R. Williams, R.F. Paige, F.A.C. Polack, and K.J. Fernandes. Concordance: An efficient framework for managing model integrity. In T. Kühne, B. Selic, M-P. Gervais, and F. Terrier, editors, *Proc. European Conference on Modelling Foundations and Applications*, volume 6138 of *Lecture Notes in Computer Science*, pages 62–73. Springer, 2010.

[Rose *et al.* 2010d] L.M. Rose, D.S. Kolovos, R.F. Paige, and F.A.C. Polack. Migrating activity diagrams with Epsilon Flock. In *Proc. Transformation Tools Contest (TTC), co-located with the International Conference on Objects, Models, Components and Patterns (TOOLS Europe)*, 2010.

[Rose *et al.* 2010e] L.M. Rose, D.S. Kolovos, R.F. Paige, and F.A.C. Polack. Model migration case. In *Proc. Transformation Tools Contest (TTC), co-located with the International Conference on Objects, Models, Components and Patterns (TOOLS Europe)*, 2010.

[Rose *et al.* 2010f] L.M. Rose, D.S. Kolovos, R.F. Paige, and F.A.C Polack. Model migration with Epsilon Flock. In L. Tratt and M. Gogolla, editors, *Proc. International Conference on the Theory and Practice of Model Transformations (ICMT)*, volume 6142 of *Lecture Notes in Computer Science*, pages 184–198. Springer, 2010.

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

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

[Sendall & Kozaczynski 2003] S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003.

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

[Sommerville 2006] I. Sommerville. *Software Engineering*. Addison-Wesley, Boston, Massachusetts, 9th edition, 2006.

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

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

[Stahl *et al.* 2006] T. Stahl, M. Voelter, and K. Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, Chichester, 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 Inc., New York, 1990.

[Steel & Raymond 2001] J. Steel and K. Raymond. Generating human-usable textual notations for information models. In *Proc. International Conference on Enterprise Distributed Object Computing (EDOC)*, pages 250–261. IEEE Computer Society, 2001.

[Steinberg *et al.* 2008] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, Boston, Massachusetts, 2008.

[Su *et al.* 2001] H. Su, D. Kramer, L. Chen, K.T. Claypool, and E.A. Rundensteiner. XEM: Managing the evolution of XML documents. In K. Aberer and L. Liu, editors, *Proc. International Workshop on Research Issues in Data Engineering (RIDE)*, pages 103–110. IEEE Computer Society, 2001.

[Tisi *et al.* 2009] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin. On the use of higher-order model transformations. In R.F. Paige, A. Hartman, and A. Rensink, editors, *Proc. European Conference on Model-Driven Architecture - Foundations and Applications (ECMDA-FA)*, volume 5562 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2009.

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

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

[Vries & Roddick 2004] D. de Vries and J.F. Roddick. Facilitating database attribute domain evolution using meso-data. In S. Wang, D. Yang, K. Tanaka, F. Grandi, S. Zhou, E.E. Mangina, T.W. Ling, I-Y. Song, J. Guan, and H.C. Mayr, editors, *Proc. Conceptual Modeling for Advanced Application Domains, ER 2004 Workshops CoMoGIS, COMWIM, ECDM, CoMoA, DGOV, and ECOMO*, volume 3289 of *Lecture Notes in Computer Science*, pages 429–440. Springer, 2004.

[W3C 2007] W3C. W3C XML Schema 1.1 Specification [online]. [Accessed 02 November 2010] Available at: `http://www.w3.org/XML/Schema`, 2007.

[Wachsmuth 2007] G. Wachsmuth. Metamodel adaptation and model co-adaptation. In E. Ernst, editor, *Proc. European Conference on Object-Oriented Programming (ECOOP)*, volume 4609 of *Lecture Notes in Computer Science*, pages 600–624. Springer, 2007.

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

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

[Watson 2008] A. Watson. A brief history of MDA. *Upgrade*, 9(2):7–11, 2008.

[Welch & Barnes 2005] P.H. Welch and F.R.M. Barnes. Communicating mobile processes. In A.E. Abdallah, C.B. Jones, and J.W. Sanders, editors, *Proc. Symposium on the Occasion of 25 Years Communicating Sequential Processes (CSP)*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer, 2005.

[Winkler & Pilgrim 2010] S. Winkler and J. von Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Software and System Modeling*, 9:529–565, 2010.