Evolution in Model-Driven Engineering

Louis M. Rose

March 5, 2010

Contents

Chapter 5

Implementation

Section 4.3 identified requirements for structures and processes for managing co-evolution. In this chapter, the way in which this thesis approaches those requirements is described. Several related solutions were implemented, using domain-specific languages, automation and extensions to existing modelling technologies. Figure 5.1 summarises the structure of this chapter. To better support co-evolution and to overcome restrictions with existing modelling frameworks, a metamodel-independent syntax was devised and implemented, enabling model and metamodel decoupling and consistency checking (Section 5.1). To address some of the challenges faced in user-driven co-evolution, an OMG specification for an alternative, textual modelling notation was implemented (Section 5.2). Model migration languages were identified, analysed and compared, leading to the derivation and implementation of a new model transformation language tailored for model migration and centred around a novel approach to relating source and target model elements (Section 5.3).

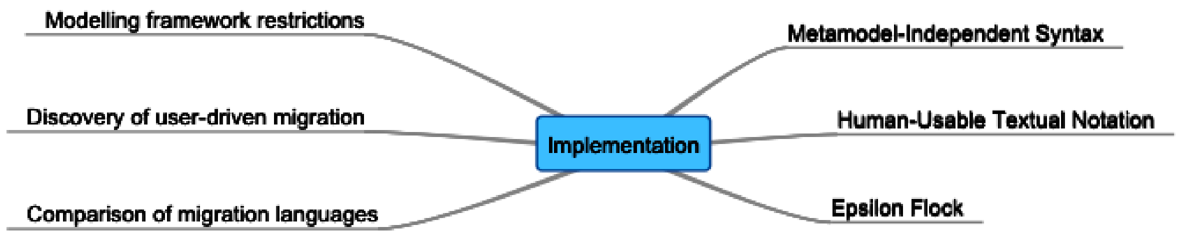

Modelling framework restrictions
Discovery of user-driven migration
Comparison of migration languages
Implementation
Metamodel-Independent Syntax
Human-Usable Textual Notation
Epsilon Flock

Figure 5.1: Implementation chapter overview.

5.1 Metamodel-Independent Syntax

Section 4.2.1 discussed the way in which modelling frameworks implicitly enforce conformance. Because of this, modelling frameworks cannot be used to load non-conformant models, and provide little support for checking the conformance of a model with other metamodels or other versions of a metamodel. In Section 4.3, these concerns lead to the identification of the following requirement: *This thesis must investigate the extension of existing modelling frameworks to support the loading of non-conformant models and conformance checking of models against other metamodels.*

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

5.1.1 Model Storage Representation

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

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

As discussed in Section 4.2.1, modelling frameworks bind a model to its metamodel using the underlying programming language. The metamodel defines the way in which model elements will be bound, and frequently, binding is strongly-typed: each metamodel type is mapped to a corresponding type in the underlying programming language. When a metamodel changes, binding may fail when a model no longer conforms to its metamodel, as the following example demonstrates.

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

```
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 5.1: Exemplar person model in XMI

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

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

5.1.2 Binding to a generic metamodel

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

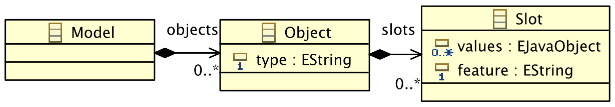


Figure 5.2: A generic metamodel.

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

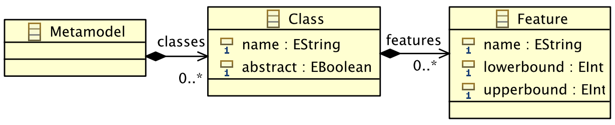


Figure 5.3: Minimal MOF metamodel.

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

1. Each object's type must be the name of some metamodel class.
2. Each object's type must be the name of some non-abstract metamodel class.
3. Each object must specify a slot for each mandatory feature of its type.
4. Each slot's feature must be the name of a metamodel feature. That metamodel feature must belong to the slot's owning object's type.
5. Each slot must be multiplicity-compatible with its feature. More specifically, each slot must contain at least as many values as its feature's lowerbound, and at most as many values as its feature's upperbound.

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

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

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

5.1.3 Example

By binding a model not to the underlying programming languages types defined in its metamodel but to the generic metamodel presented in Figure 5.2, conformance can be checked using the above constraints. Binding the exemplar XMI in Listing 5.1 to the generic metamodel shown in Figure 5.2 produces three Objects, all with type “Person”. Each class object contains a slot whose feature is name, one with the value “Franz”, one with the value “Julie” and the other with the value “Hermann”. The object containing the slot with value “Franz” contains two further slots: one whose feature is mother and whose value is a reference[1] to the object that contains the name slot with the value “Julie” and one whose feature is father and whose value is a reference to the object containing “Hermann”. A UML object diagram for this instantiation of the generic metamodel is shown in Figure 5.4. Instances of object (slot) are shaded grey (white).

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

5.1.4 Applications

As this section has shown, binding to a metamodel independent syntax is an alternative model deserialisation mechanism that can be used when a model no longer conforms to its metamodel and to check the conformance of a model with any metamodel. The metamodel independent syntax described in this section

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

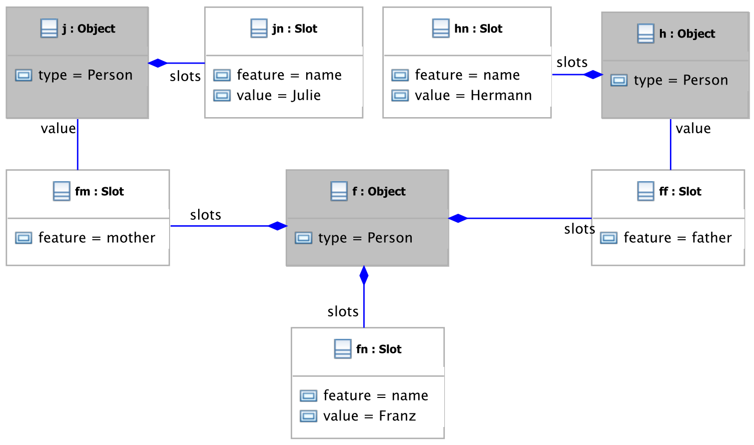


Figure 5.4: Exemplar instantiation of generic metamodel.

is used throughout this chapter to support other structures and processes for co-evolution.

In Section 5.2, a textual modelling notation is integrated with the metamodel independent model representation discussed here. In Section 5.3, a domain-specific language for migration uses metamodel independent syntax to perform partial migration by producing models that conform to a generic metamodel rather than their evolved metamodel.

Automatic Consistency Checking

In addition to the applications outlined above, a metamodel independent syntax is particularly useful during metamodel installation. As discussed in Section 4.2.1, metamodel developers do not have access to downstream models. Consequently, instances of a metamodel may become non-conformant after a new version of a metamodel plug-in is installed. By default, an EMF metamodel plug-in does not check conformance during plug-in installation and non-conformant models are only detected when the user attempts to load them.

To enable conformance checking as part of metamodel installation in EMF, the binding to a generic metamodel discussed above has been integrated with Concordance [Rose *et al.* 2010a] in joint work with Dimitrios S. Kolovos, a lecturer in this department, Nicholas Drivalos, a research associate in this department and James R. Williams, a research student in this department[2].

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

The integration work involved extending Concordance such that, after the installation of a metamodel plug-in, models that conform to any previous ver-

[2]TODO: Ask R+F: do I need to say something like “No credit is claimed for this joint work.”

sion of the installed metamodel are identified. Those models are checked for conformance with the new metamodel. As such, conformance checking occurs automatically and during metamodel installation. Conformance problems are detected and reported immediately, rather than when the user next attempts to load an affected model. By integrating conformance checking with Concordance, increased scalability is achieved, as demonstrated in [Rose *et al.* 2010a].

5.2 Textual Modelling Notation

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

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

5.2.1 Human-Usable Textual Notation

The OMG's Human-Usable Textual Notation (HUTN) [OMG 2004] defines a textual modelling notation, which aims to conform to human-usability criteria [OMG 2004]. There is no current reference implementation of HUTN: the Distributed Systems Technology Centre's TokTok project (an implementation of the HUTN specification) is inactive (and the source code can no longer be found), whilst work on implementing the HUTN specification by Muller and Hassenforder [Muller & Hassenforder 2005] has been abandoned in favour of Sintaks [IRISA 2007], which operates upon domain-specific concrete syntax.

Model storage representations are often optimised to reduce storage space or to increase the speed of random access, rather than for human usability. By contrast, the HUTN specification states its primary design goal as human-usability and "this is achieved through consideration of the successes and failures of common programming languages" [OMG 2004, Section 2.2]. The HUTN specification refers to two studies of programming language usability to justify design decisions. Because no reference implementation exists, the specification does not evaluate the human-usability of the notation. This thesis proposes that

HUTN be used instead of XMI for user-driven co-evolution. Further discussion of the human-usability of HUTN is deferred to Chapter 6.

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

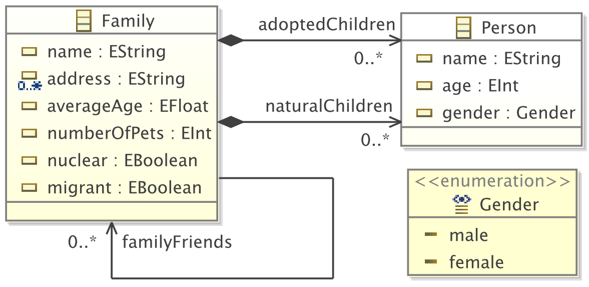


Figure 5.5: Exemplar families metamodel. (Shading is irrelevant).

Basic Notation

Listing 5.2 shows the construction of an *object* in HUTN, here an instance of the Family class from Figure 5.5. Line 1 specifies the package containing the classes to be constructed (`FamilyPackage`) and a corresponding identifier (`families`), used for fully-qualifying references to objects (Section 5.2.1). Line 2 names the class (`Family`) and gives an identifier for the object (`The Smiths`). Lines 3 to 7 define *attribute values*; in each case, the data value is assigned to the attribute with the specified name. The encoding of the value depends on its type: strings are delimited by any form of quotation mark; multi-valued attributes use comma separators, etc.

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

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

Listing 5.2: Specifying attributes with HUTN.

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

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

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

Listing 5.4: Specifying a simple reference with HUTN.

Keywords and Adjectives

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

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

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

Listing 5.5: Using keywords and adjectives in HUTN.

Inter-Package References

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

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

Listing 5.6: Referencing objects in other packages with HUTN.

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

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

Alternative Reference Syntax

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

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

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

Listing 5.7: Using a reference block in HUTN.

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

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

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

Listing 5.8: Using an infix reference in HUTN.

Customisation via Configuration

Some limited customisation of HUTN for particular metamodels can be achieved using *configuration files*. Customisations permitted include a parametric form of object instantiation (not yet implemented); renaming of metamodel elements; giving default values for attributes; and stating an attribute whose values are used to infer a default identifier.

5.2.2 Epsilon HUTN

To investigate the extent to which it can be used during user-driven co-evolution, an implementation of HUTN was constructed. The implementation, Epsilon HUTN, makes extensive use of the Epsilon model management platform. Before presenting HUTN, it is necessary to revisit some details of the Epsilon [Kolovos 2009] platform, which was introduced in Section 3.1.1.

The Epsilon Platform

Epsilon, a component of the Eclipse GMT project [Eclipse 2008], provides infrastructure for implementing uniform and interoperable model management languages, for performing tasks such as model merging, model transformation and inter-model consistency checking.

The core of the platform is the Epsilon Object Language (EOL) [Kolovos *et al.* 2006], a reworking and extension of OCL that includes the ability to update models, conditional and loop statements, statement sequencing, and access to standard I/O streams. EOL provides mechanisms for reusing sections of code, such as user-defined operators along with modules and import statements. The Epsilon task-specific languages are built atop EOL, giving highly efficient inheritance and reuse of features. Currently, these task-specific languages include support for model-to-model transformation (ETL [Kolovos *et al.* 2008a]), model-to-text transformation (EGL [Rose *et al.* 2008]) and model validation (EVL [Kolovos *et al.* 2008b]).

Implementation of Epsilon HUTN

Epsilon HUTN uses the task-specific 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. Epsilon HUTN has been released as part of Epsilon, and includes development tools for Eclipse.

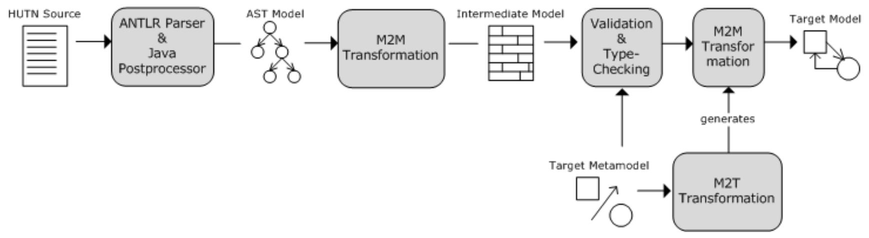


Figure 5.6: The architecture of Epsilon HUTN.

Figure 5.6 outlines the workflow through Epsilon HUTN, from HUTN source text to instantiated target model. The HUTN model specification is parsed to an abstract syntax tree using a HUTN parser specified in ANTLR [Parr 2007]. From this, a Java postprocessor is used to construct an instance of a simple AST metamodel (which comprises two meta-classes, Tree and Node). Using ETL, M2M transformations are then applied to produce an instance of the generic metamodel discussed in Section 5.1. Finally, a M2T transformation on the target metamodel, specified in EGL, produces a further M2M transformation, from the generic metamodel to the target model.

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

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

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

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

```
1  rule NameNode2PackageObject
2     transform n : AntlrAst!NameNode
3     to p : Intermediate!PackageObject {
4
5     guard : n.parent.isUndefined()
6
7     p.type := n.text;
8     p.line := n.line;
9     p.col := n.column;
10
11    var slot := new Intermediate!ContainmentSlot;
12    for (child in n.children) {
13       slot.objects.add(child.equivalent());
14    }
15    if (slot.objects.notEmpty()) {
16       p.slots.add(slot);
17    }
}
```

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

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

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

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

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

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

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

Listing 5.11 is the EGL template for a M2T transformation on the target metamodel; it generates the M2M transformation used for generating the target model. The loop beginning on line 1 iterates over each meta-class in the metamodel, producing a transformation rule to generate target model instances of that meta-class from class objects in the intermediate model. The template guard (line 6) specifies that only class objects of the same type as the meta-class be transformed by the current rule. The template body is omitted from Listing 5.11 – it 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.

```
[% for (class in EClass.allInstances()) { %]
rule Object2[%=class.name%]
  transform o : Intermediate!ClassObject
  to t : Model![%=class.name%] {

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

    -- body omitted for brevity
  }
[% } %]
```

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

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

5.2.3 Migration with HUTN

Epsilon HUTN uses the generic metamodel (from Section 5.1) as an intermediary, facilitating transformation from XMI to HUTN (i.e. the inverse of the transformation discussed above): XMI is parsed to produce an instance of the generic metamodel, and an unparser (implemented using the visitor design pattern) generates HUTN source. In this manner, HUTN can be generated for any XMI document, regardless of whether the model described by the XMI conforms to its metamodel.[3]

To demonstrate the way in which HUTN can be used to perform migration, the exemplar XMI shown in Listing 5.1 is represented using HUTN in Listing 5.12. Recall that the XMI describes three Persons, Franz, Julie and Hermann. Julie and Hermann are the mother and father of Franz.

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

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

Listing 5.12: Exemplar HUTN document.

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 **??**.

Resolving the conformance problems requires the user to change the feature named in the infix associations from mother (father) to parents. The Epsilon HUTN development tools provide content assistance, which might be useful in this situation. Listing 5.13 shows a HUTN document that conforms to the metamodel defining parents rather than mother and father.

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

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

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

Listing 5.13: Exemplar HUTN document.

5.2.4 Limitations

TODO: Ask Richard and Fiona whether the following discussion of HUTN's limitations might be better placed in the Evaluation chapter.

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:

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

A failure behaviour can contain many expressions, and each component may be connected to many other components, so the metamodel for failure behaviours contains a large number of classes. In the generic concrete syntax, the specification of these behaviours is unhelpfully terse. For example. Listing 5.14 gives the HUTN syntax for failure behaviour (5.1), above.

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

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

Listing 5.14: Failure behaviour specified in HUTN.

In general, HUTN is less concise than a domain-specific syntax for metamodels containing a large number of classes with few attributes, and in cases where most attributes are used to define structural relationships among concepts. However, there might still be benefits from using HUTN in such cases, if the metamodel is likely to be modified frequently, of it the model does not yet have a formal metamodel.

5.2.5 Summary

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

5.3 Epsilon Flock

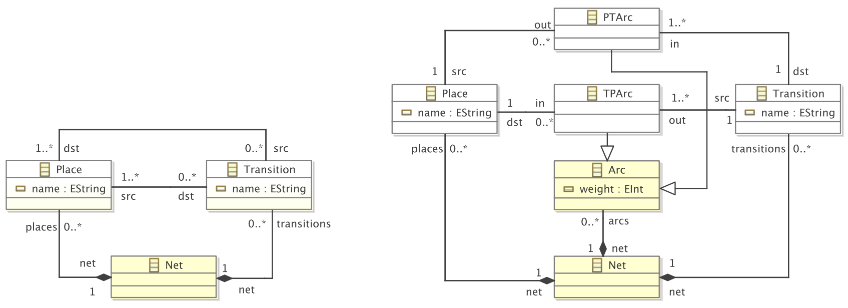
Section 4.2.3 discussed existing approaches to model migration, highlighting variation in the languages used for specifying migration strategies. In this section, migration strategy languages are compared, using the example of metamodel evolution given in Section 5.3.1. From this comparison, requirements for a domain-specific language for specifying and executing model migration strategies are derived (Section 5.3.3) and an implementation is described (Section 5.3.4). This work described in this section was published in [Rose *et al.* 2010b].

5.3.1 Co-Evolution Example

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

In Figure 5.7(a), a Petri `Net` comprises `Places` and `Transitions`. A `Place` has any number of `src` or `dst` `Transitions`. Similarly, a `Transition` has at least one `src` and `dst` `Place`. In this example, the metamodel in Figure 5.7(a) is to be evolved so as to support weighted connections between `Places` and `Transitions` and between `Transitions` and `Places`.

The evolved metamodel is shown in Figure 5.7(b). `Places` are connected to `Transitions` via instances of `PTArc`. Likewise, `Transitions` are connected



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

Figure 5.7: Exemplar metamodel evolution. (Shading is irrelevant). Taken from [Rose *et al.* 2010b].

to `Places` via `TPArc`. Both `PTArc` and `TPArc` inherit from `Arc`, and therefore can be used to specify a `weight`.

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

```
1. For every instance, t, of Transition:
       For every Place, s, referenced by the src feature of t:
           Create a new instance, arc, of PTArc.
           Set s as the src of arc.
           Set t as the dst of arc.
           Add arc to the arcs reference of the Net referenced by t.
       For every Place, d, referenced by the dst feature of t:
           Create a new instance, arc, of TPArc.
           Set t as the src of arc.
           Set d as the dst of arc.
           Add arc to the arcs reference of the Net referenced by t.
2. And nothing else changes.
```

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

5.3.2 Existing Approaches

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

Manual Specification with Model-to-Model Transformation

A model-to-model transformation specified between original and evolved metamodel can be used for performing model migration. Part of the model migration for the Petri nets metamodel is codified with the Atlas Transformation Language (ATL) [Jouault & Kurtev 2005] in Listing 5.15. Rules for migrating `Places` and `TPArcs` have been omitted for brevity, but are similar to the `Nets` and `PTArcs` rules.

In ATL, *rule*s transform source model elements (specified using the `from` keyword) to target model elements (specified using `to` keyword). For example, the `Nets` rule on line 1 of Listing 5.15 transforms an instance of `Net` from the original (source) model to an instance of `Net` in the evolved (target) model. The source model element (the variable `o` in the `Net` rule) is used to populate the target model element (the variable `m`). ATL allows rules to be specified as *lazy* (not scheduled automatically and applied only when called by other rules).

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

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

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

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

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

In model migration, source and target metamodels differ, and hence existing target transformations cannot be used to specify model migration strategies. Consequently, model migration strategies are specified with new target model-to-model transformation languages, and often contain sections for copying from original to migrated model those model elements that have not been affected by metamodel evolution. For the Petri nets example, the `Nets` rule (in Listing 5.15) and the `Places` rule (not shown) exist only for this reason.

The `Transitions` rule in Listing 5.15 codifies in ATL the migration strategy described previously. The rule is executed for each `Transition` in the original model, `o`, and constructs a `PTArc` (`TPArc`) for each reference to a `Place` in `o.src` (`o.dst`). Lazy rules must be used to produce the arcs to prevent circular dependencies with the `Transitions` and `Places` rules. Here, ATL, a typical rule-based transformation language, is considered and model migration would be similar in QVT. With Kermeta, migration would be specified in an imperative style using statements for copying `Nets`, `Places` and `Transitions`, and for creating `PTArcs` and `TPArcs`.

Manual Specification with Ecore2Ecore Mapping

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

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

Model migration is specified on the XMI representation of the model and hence presumes some knowledge of the XMI standard. For example, in XMI, references to other model elements are serialised as a space delimited collection of URI fragments [Steinberg *et al.* 2008]. Listing 5.16 shows a section of the Ecore2Ecore model migration for the Petri net example presented above. The method shown converts a `String` containing URI fragments to a `Collection` of `Places`. The method is used to access the `src` and `dst` features of `Transition`, which no longer exist in the evolved metamodel and hence are not loaded automatically by EMF. To specify the migration strategy for the Petri nets example, the metamodel developer must know the way in which the `src` and `dst` features are represented in XMI. The complete listing, not shown here, exceeds 200 lines of code.

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

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

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

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

    places.add((Place)eObject);
  }

  return places;
}
```

Listing 5.16: Java method for deserialising a reference.

Operator-based Co-evolution with COPE

Operator-based approaches to managing co-evolution, such as COPE [Herrmannsdoerfer *et al.* 2009], 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.* 2009] evolves the metamodel such that a non-containment reference becomes a containment reference and migrates models such that the values of the evolved reference are replaced by copies. By composing co-evolutionary operators, metamodel evolution can be performed and a migration strategy can be generated without writing any code.

To perform metamodel evolution using an operator-based approach, the library of co-evolutionary operators must be integrated with tools for editing metamodels. COPE provides integration with the EMF tree-based metamodel editor. Operators may be applied to an EMF metamodel, and a record of changes tracks their application. Once metamodel evolution is complete, a migration strategy can be generated automatically from the record of changes. The migration strategy is distributed along with the updated metamodel, and metamodel users choose when to execute the migration strategy on their models.

To be effective, operator-based approaches must provide a rich yet navigable library of co-evolutionary operators, as discussed in Section 4.2.3. To this

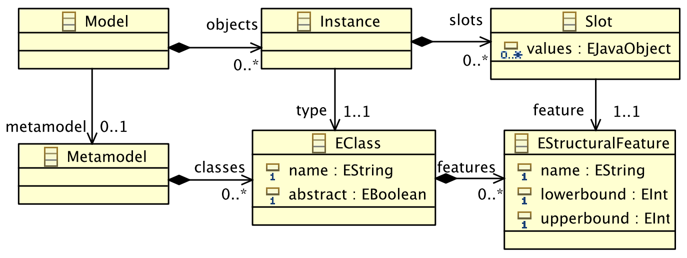


Figure 5.8: Simplification of the metamodel-independent representation used by COPE, based on [Herrmannsdoerfer *et al.* 2009].

end, COPE allows model migration strategies to be specified manually when no co-evolutionary operator is appropriate. Rather than use either of the two manual specification approaches discussed above (model-to-model transformation and Ecore2Ecore mapping), COPE employs a fundamentally different approach using an existing target transformation.

As discussed above, existing target transformations cannot be used for specifying model migration strategies as the source (original) and target (evolved) metamodels differ. However, models can be structured independently of their metamodel using a *metamodel-independent representation*. Figure 5.8 shows a simplification of the metamodel-independent representation used by COPE. By using a metamodel-independent representation of models as an intermediary, an existing target transformation can be used for performing model migration when the migration strategy is specified in terms of the metamodel-independent representation. Further details of this technique are given in [Herrmannsdoerfer *et al.* 2009].

Listing 5.17 shows the COPE model migration strategy for the Petri net example given above[4]. Most notably, slots for features that no longer exist must be explicitly `unset`. In Listing 5.17, slots are `unset` on four occasions, once for each feature that exists in the original metamodel but not the evolved metamodel. Namely, these features are: `src` and `dst` of `Transition` and of `Place`. Failing to `unset` slots that do not conform with the evolved metamodel causes migration to fail with an error.

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

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

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

Listing 5.17: Petri nets model migration in COPE

[4]In Listing 5.17, some of the concrete syntax has been changed in the interest of brevity.

5.3.3 Analysis

By analysing existing approaches to managing developer-driven co-evolution, requirements were derived for Epsilon Flock, a domain-specific language for specifying and executing model migration. The derivation of the requirements for Epsilon Flock is now summarised, by considering two dimensions: 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

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

In each case, extra effort is required when defining a migration strategy due to the way in which the co-evolution approach relates source (original) and target (migrated) model elements. This observation led to the following requirement: *Epsilon Flock must* ***automatically*** *copy every model element that conforms to the evolved metamodel from original to migrated model, and must not automatically copy any model element that does not conform to the evolved metamodel from original to migrated model.*

Model Representation

When using the Ecore2Ecore approach, model elements that do not conform to the evolved metamodel are accessed via XMI. Consequently, the metamodel developer must be familiar with XMI and must perform tasks such as dereferencing URI fragments (Listing 5.16) and type conversion. With COPE and the Epsilon Transformation Language, models are loaded using a modelling framework (and so migration strategies need not be concerned with the representation used to store models). Consequently, the following requirement was identified: *Epsilon Flock must not expose the underlying representation of original or migrated models.*

To apply co-evolution operators, COPE requires the metamodel developer to use a specialised metamodel editor, which can manipulate only metamodels defined with EMF. Like, the Ecore2Ecore approach, COPE can be used only to manage co-evolution for models and metamodels specified with EMF. Tight coupling to EMF allows the Ecore2Ecore approach to schedule migration automatically, during model loading. To better support integration with modelling frameworks other than EMF, the following requirement was derived: *Epsilon Flock must be loosely coupled with modelling frameworks and must not assume that models and metamodels will be represented in EMF.*

5.3.4 Implementation

Driven by the analysis presented above, Epsilon Flock (subsequently referred to as Flock) was designed and implemented. Flock is a domain-specific language for specifying and executing model migration strategies. Flock uses a model connectivity framework, which decouples migration from the representation of models and provides compatibility with several modelling frameworks (Section 3.1.1). Flock automatically maps each element of the original model to an equivalent element of the migrated model using a novel conservative copying algorithm and user-defined migration rules (Section 5.3.6).

5.3.5 The Epsilon Platform

Before presenting Flock, it is necessary to revisit some details of the Epsilon [Kolovos 2009] platform, which was introduced in Section 3.1.1. Epsilon, a component of the Eclipse GMT project [Eclipse 2008], provides infrastructure for implementing uniform and interoperable model management languages, for performing tasks such as model merging, model transformation and inter-model consistency checking.

The core of the platform is the Epsilon Object Language (EOL) [Kolovos *et al.* 2006], a reworking and extension of OCL that includes the ability to update models, conditional and loop statements, statement sequencing, and access to standard I/O streams. EOL provides mechanisms for reusing sections of code, such as user-defined operators along with modules and import statements. The Epsilon task-specific languages are built atop EOL, giving highly efficient inheritance and reuse of features.

5.3.6 Flock

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

Abstract Syntax

As illustrated by Figure 5.9, Flock migration strategies are organised into modules (`FlockModule`), which inherit from EOL modules (`EolModule`), which provides support for module reuse with import statements and user-defined operations. Modules comprise any number of rules (`Rule`). Each rule has an original metamodel type (`originalType`) and can optionally specify a `guard`, which is either an EOL statement or a block of EOL statements. `MigrateRules` must specify an evolved metamodel type (`evolvedType`) and/or a `body` comprising a block of EOL statements.

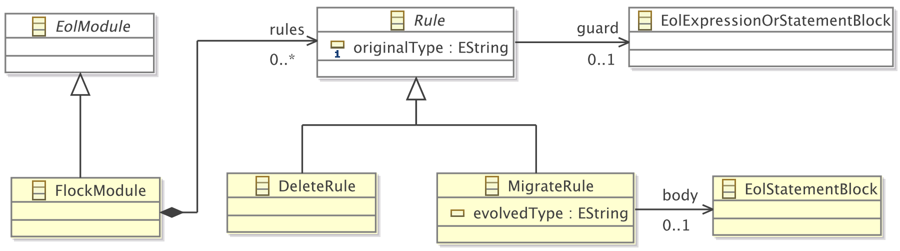


Figure 5.9: The abstract syntax of Flock.

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

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

Listing 5.18: Concrete syntax of migrate and delete rules.

Concrete Syntax

Listing 5.18 shows the concrete syntax of migrate and delete rules. All rules begin with a keyword indicating their type (either `migrate` or `delete`), followed by the original metamodel type. Guards are specified using the `when` keywords. Migrate rules may also specify an evolved metamodel type using the `to` keyword and a `body` as a (possibly empty) sequence of EOL statements.

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

Execution Semantics

A Flock module has the following behaviour when executed:

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

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

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

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

   Do nothing.

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

Create an equivalent model element, `e'` in the migrated model. The metaclass of `e'` is determined from the `evolvedType` (or the `originalType` when no `evolvedType` has been specified) of `r`.

Copy the data contained in `e` to `e'` (using the *conservative copy* algorithm described in the sequel).

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

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

Conservative Copying

Flock contributes an algorithm, termed *conservative copy*, that copies model elements from original to migrated model only when those model elements conform to the evolved metamodel. Because of its conservative copy algorithm, Flock is a hybrid of new target and existing target transformation languages. This section discusses the conservative copying algorithm in more detail.

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

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

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

  When no equivalent metafeature can be found, do nothing.

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

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

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

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

Flock implements conservative copy by delegate conformance checking responsibilities to EMC.

Finally, some categories of model value must be converted before being copied from the original to the migrated model. Again, the need for and semantics of this conversion varies over modelling frameworks. Reference values typically require conversion before copying. In this case, the mappings between original and migrated model elements maintained by the Flock runtime can be used to perform the conversion. In other cases, the target modelling framework must be used to perform the conversion, such as when EMF enumeration literals are to be copied.

Development and User Tools

As discussed in Section 4.2, models and metamodels are typically kept separate. Flock migration strategies can be distributed by the metamodel developer in two ways. An extension point defined by Flock provides a generic user interface for migration strategy execution. Alternatively, metamodel developers can elect to build their own interface, delegating execution responsibility to `FlockModule`. We anticipate the latter to be useful for production environments using model or source code management repositories.

5.3.7 Example

The exemplar Petri net metamodel evolution is now revisited to demonstrate the basic functionality of Flock. In Listing 5.19, `Nets` and `Places` are migrated automatically. Unlike the ATL migration strategy (Listing 5.15), no explicit copying rules are required. Compared to the COPE migration strategy (Listing 5.17), the Flock migration strategy does not explicitly unset the original `src` and `dst` features of `Transition`.

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

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

Listing 5.19: Petri nets model migration in Flock

Table 5.1 illustrates several characterising differences between Flock and the related approaches presented in Section 5.3.1. Due to its conservative copying algorithm, Flock is the only approach to provide both automatic copying and unsetting. Automatic copying is significant for metamodel evolutions with a large number of unchanging features.

All of the approaches considered in Table 5.1 support EMF, arguably the most widely used modelling framework. The Ecore2Ecore approach, however, requires migration to be encoded at the level of the underlying model representation XMI. Both Flock and ATL support other modelling technologies, such as MDR and XML. However, ATL does not automatically copy model elements that have not been affected by metamodel changes. Therefore, migration between models of different technologies with ATL requires extra statements in the migration strategy to ensure that the conformance constraints of the target technology are satisfied. Because it delegates conformance checking to an EMC driver, Flock requires no such checks.

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

5.4 Chapter Summary

To be completed.

Table 5.1: Properties of model migration approaches

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

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[Rose *et al.* 2010b] Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Model migration with flock. In *In preparation*, 2010.

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

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

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

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

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

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

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