Model Migration with Epsilon Flock

Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack

Department of Computer Science, University of York, UK.
[louis,dkolovos,paige,fiona]@cs.york.ac.uk

Abstract. In their recent book, Mens and Demeyer state that Model-Driven Engineering introduces additional challenges for controlling and managing software evolution. Today, tools exist for generating model editors and for managing models with transformation, validation, merging and weaving. There is limited support, however, for *model migration* - a development activity in which instance models are updated in response to metamodel evolution. In this paper, we describe Epsilon Flock, a model-to-model transformation language tailored for model migration that contributes a novel algorithm for relating source and target model elements. To demonstrate its conciseness, we compare Flock to other approaches.

1 Introduction

Today, metamodel developers can automatically generate tools [20] and graphical editors [6] for manipulating models. Models can be *managed* using a variety of operations, such as: transformation to other models [9], transformation to text [15], and validation against a set of constraints. There is limited support, however, for *model migration* - a development activity in which instance models are updated in response to metamodel evolution. More generally, MDE introduces additional challenges for controlling and managing software evolution [13].

When a metamodel evolves, instance models might no longer conform to the structures and rules defined by the metamodel. When an instance model does not conform to its metamodel, it cannot be manipulated with metamodel-specific editors, cannot be managed with model management operations and, in some cases, cannot be loaded by modelling tools.

In this paper, we compare existing approaches for model migration and perform a gap analysis in Section 3. From this analysis, we have derived Epsilon Flock, a model-to-model transformation language tailored for model migration (Section 4). Epsilon Flock contributes several novel features including: a hybrid approach to relating source and target model elements, migration between models specified in heterogenous modelling technologies, and a more concise syntax than existing approaches. In Section 5, we apply Epsilon Flock to two examples of co-evolution for comparison with existing approaches.

2 Background

Before introducing existing approaches, a more thorough definition of model migration is presented in this section, along with a discussion of some of the

characteristics of MDE modelling frameworks that affect the way in which model migration can be performed.

2.1 Conformance

A model *conforms to* a metamodel when the metamodel specifies every concept used in the model definition, and the model uses the metamodel concepts according to the rules specified by the metamodel. Conformance can be described by a set of constraints between models and metamodels [17]. When all constraints are satisfied, a model conforms to a metamodel. For example, a conformance constraint might state that every object in the model has a corresponding non-abstract class in the metamodel.

Metamodel changes can affect conformance. For example, when a concept is removed from a metamodel, any models using that concept no longer conform to the metamodel. *Model and metamodel co-evolution* (subsequently referred to as *co-evolution*) is the process of evolving model and metamodel such that conformance is preserved. *Model migration* is a development activity in which models are updated in response to metamodel evolution to re-establish conformance.

2.2 Relevant Characteristics of MDE Modelling Frameworks

Model and Metamodel Separation In modern MDE development environments, models and metamodels are kept separate. Metamodels are developed and distributed to users. Metamodels are installed, configured and combined to form a customised MDE development environment. Metamodel developers have no programmatic access to downstream instance models.

Because of this, metamodel evolution occurs independently to model migration. First, the metamodel is evolved. Subsequently, the users of the metamodel find that their models are out-of-date and migrate their models. This process is facilitated when, during metamodel evolution, the metamodel developer devises and codifies a *migration strategy*, which is distributed with the evolved metamodel. Later, the metamodel user executes the migration strategy to migrate models that no longer conform to the metamodel. When no migration strategy is included with an evolved metamodel, model migration is a tedious and error-prone process, as we discuss in [19].

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

3 Existing Approaches

We now compare existing approaches to managing co-evolution, using the example of metamodel evolution given in Section 3.1. From this comparison, we derive requirements for a domain-specific language for specifying and executing model migration strategies in Section 3.5.

In [18], we propose three categories of co-evolution approaches. In *manual specification*, migration strategies are specified by hand. In *operator-based* approaches, operators are used to evolve a metamodel and then to derive a corresponding migration strategy. In *metamodel matching*, the original and evolved metamodels are compared, and their differences used to generate a migration strategy.

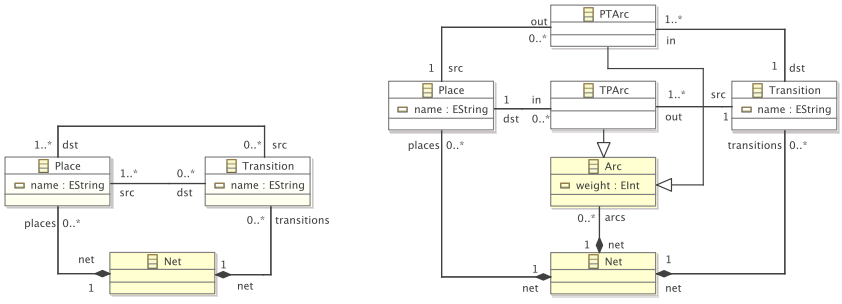
We know of no real world projects that have managed co-evolution with metamodel matching approaches and, to date, only prototypical metamodel matching approaches exist [1, 5]. Furthermore, metamodel matching approaches cannot always automatically infer an appropriate migration strategy [18]. For these reasons, metamodel matching approaches are not considered further in this report.

Instead, we discuss co-evolution approaches that have been used in projects employing MDE. Model-to-model transformation (Section 3.2) and use of an Ecore2Ecore mapping (Section 3.3) are manual specification approaches. The former has been used in the Eclipse GMF project [6] and the latter in the Eclipse MDT UML2 project [4]. Section 3.4 discusses COPE, an operator-based approach, which has been applied to real world projects [7].

3.1 Co-Evolution Example

In this paper, we use the example of an evolution of a Petri net metamodel, previously used in [1, 5, 21] to discuss co-evolution and model migration.

In Figure 1(a), a Petri `Net` comprises `Places` and `Transitions`. A `Place` has any number of `src` or `dst` `Transitions`. Similarly, a `Transition` has at



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

Fig. 1. Exemplar metamodel evolution. (Shading is irrelevant).

least one src and dst Place. In this example, the metamodel in Figure 1(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 1(b). Places are connected to Transitions via instances of PTArc. Likewise, Transitions are connected to Places via TPArc. Both PTArc and TPArc inherit from Arc, and therefore can be used to specify a weight.

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

1. For every instance, t, of Transition:
   - For every Place, s, referenced by the src feature of t:
   -- Create a new instance, arc, of PTArc.
   -- 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.

Step 2 is necessary to highlight that migration should change only those model elements that have been affected by the metamodel evolution. Unaffected model elements should not be changed by the migration strategy. Using the above example, the existing approaches for specifying and executing model migration strategies are now compared.

3.2 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. This section briefly discusses two styles of model-to-model transformation and presents an example of a migrating model-to-model transformation.

In model transformation, [2] identifies two categories of relationship between source and target model, *new-target* and *existing-target*. In the former, the target model is constructed entirely by the transformation. In the latter, the target model is initialised as a copy of the source model before the transformation.

In model migration, source and target metamodels differ, and hence existing-target transformations cannot be used. 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.

Part of the Petri nets model migration is codified with the Atlas Transformation Language (ATL) [9] in Listing 1.1. 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 1.1 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* (applied only when called by other rules).

The `Transitions` rule in Listing 1.1 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. On line 10, the feature `in` is escaped because `in` is an ATL keyword.

```
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 1.1. Fragment of the Petri nets model migration in ATL.

As discussed above, a new-target transformation must be used to specify migration because the source and target metamodels differ. For the Petri nets example, the `Nets` rule (in Listing 1.1) and the `Places` rule (not shown) exist only to copy data from the original to the migrated model.

Here, we have considered ATL, which is a typical rule-based transformation language. Model migration would be similar in QVT [16]. With Kermeta [14], migration would be specified in an imperative style using statements for copying `Nets`, `Places` and `Transitions`, and for creating `PTArcs` and `TPArcs`.

3.3 Manual Specification with Ecore2Ecore Mapping

Hussey and Paternostro [8] explain the way in which integration with the model loading mechanisms of the Eclipse Modeling Framework (EMF) [20] 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 2.2), 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 [20]. For the Petri net example presented above, the Ecore2Ecore migration strategy must 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 do this, the Ecore2Ecore migration strategy must convert a `String` containing URI fragments to a `Collection` of `Places`. In other words, 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 Ecore2Ecore migration strategy for the Petri nets example, not shown here, exceeds 200 lines of code.

3.4 Operator-based Co-evolution with COPE

Operator-based approaches to managing co-evolution, such as COPE [7], 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 [7] 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 we discuss in [18]. To this end, COPE allows model migration strategies to be specified manually when no co-evolutionary operator is appropriate. Rather than use either of the two manual specification approaches discussed above (model-to-model transformation and

Ecore2Ecore mapping), COPE employs a fundamentally different approach for manually specifying migration strategies 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*. 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 [7].

Listing 1.2 shows the COPE model migration strategy for the Petri net example given above[1]. Most notably, slots for features that no longer exist must be explicitly `unset`. In Listing 1.2, 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.

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

Listing 1.2. Petri nets model migration in COPE

3.5 Analysis

By analysing the above approaches to managing 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 the way in which languages used for specifying migration strategies relate source and target elements and represent models.

Source-Target Relationship New target transformation languages (Section 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

[1] In Listing 1.2, some of the concrete syntax has been changed in the interest of brevity.

contrast, model migration strategies written in COPE (Section 3.4) must explicitly unset any data that is not to be copied from the original to the migrated model. The Ecore2Ecore approach (Section 3.3) 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 and type conversion. With COPE and the Atlas 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.*

4 Epsilon Flock

Driven by the analysis presented in Section 3, we have designed and implemented Epsilon Flock (subsequently referred to as Flock). 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 4.1). 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 4.2).

4.1 The Epsilon Platform

Before presenting Flock, it is necessary to introduce the underlying Epsilon [10] platform. Epsilon, a component of the Eclipse GMT project [3], 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) [11], 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.

4.2 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 [9] and the Epsilon Transformation Language [12]. 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 2, 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.

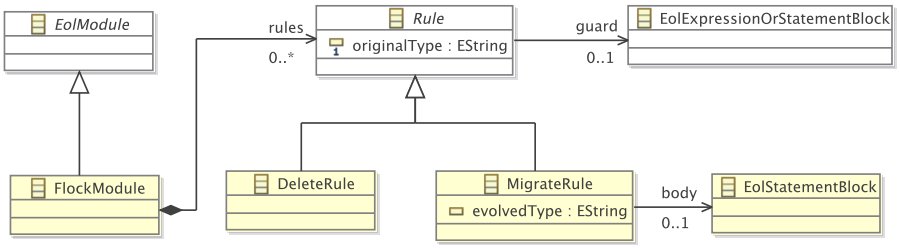


Fig. 2. The abstract syntax of Flock.

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

Listing 1.3. Concrete syntax of migrate and delete rules.

Concrete Syntax Listing 1.3 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.
   – When more than one rule could be applied, the first to appear in the Flock source file is selected.
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- 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 2.2, 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 Example

Flock is now demonstrated using two examples of model migration. Listing 1.4 illustrates the Flock migration strategy for the Petri net example introduced in Section 3, which is included for direct comparison with other approaches. In addition, we present a larger example, based on changes between versions 1.5 and 2.0 of the UML specification.

5.1 Petri Nets in Flock

In Listing 1.4, `Nets` and `Places` are migrated automatically. Unlike the ATL migration strategy (Listing 1.1), no explicit copying rules are required. Compared to the COPE migration strategy (Listing 1.2), the Flock migration strategy does not explicitly unset the original `src` and `dst` features of `Transition`.

```
1  migrate Transition {
2    for (source in original.src) {
3      var arc := new Migrated!PTArc;
4      arc.src := source.equivalent(); arc.dst := migrated;
5      arc.net := original.net.equivalent();
6    }
7
8    for (destination in original.dst) {
9      var arc := new Migrated!TPArc;
10     arc.src := migrated; arc.dst := destination.equivalent();
11     arc.net := original.net.equivalent();
12   }
13 }
```

Listing 1.4. Petri nets model migration in Flock

```
1  migrate Association {
2    migrated.memberEnds := original.connections.equivalent();
3  }
4
5  migrate Class {
6    var fs := original.features.equivalent();
7    migrated.operations := fs.select(f|f.isKindOf(Operation));
8    migrated.attributes := fs.select(f|f.isKindOf(Property));
9    migrated.attributes.addAll(original.associations.equivalent())
10 }
11
12 delete StructuralFeature when: original.targetScope <> #instance
13
14 migrate Attribute to Property {
15   if (original.ownerScope = #classifier) {
16     migrated.isStatic = true;
17   }
```

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

Listing 1.5. UML model migration in Flock

5.2 UML 1.5 to UML 2.0 in Flock

Figure 3 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 3(a), association ends and attributes are specified explicitly and separately. In Figure 3(b), the `Property` class is used instead. The Flock migration strategy (Listing 1.5) for Figure 3 is now discussed.

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

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

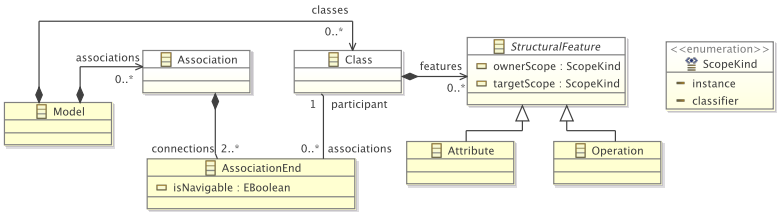
The UML 1.5 `StructuralFeature#targetScope` feature is no longer supported in UML 2.0, and no migration path is provided. Consequently, line 14 deletes any model element whose `targetScope` is not the default value.

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

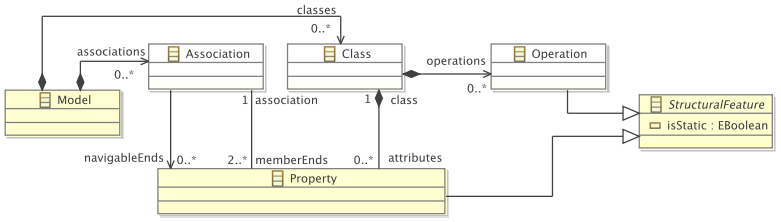
5.3 Comparison

Table 1 illustrates several characterising differences between Flock and the related approaches presented in Section 3. 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, such as those observed for UML.

All of the approaches considered in Table 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



(a) Original metamodel.



(b) Evolved metamodel.

Fig. 3. Exemplar UML metamodel evolution. (Shading is irrelevant).

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.

6 Conclusions and Further Work

Existing approaches for managing model and metamodel co-evolution in the context of model-driven engineering have been reviewed. The way in which existing approaches either require the copying of model elements from the original to the migrated model or the deletion of model elements from the migrated model has been discussed. To this end, the design and implementation of Epsilon Flock, a

Table 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

model-to-model transformation language tailored for model migration, has been presented. Flock treats the relationship between source (original) and target (migrated) model elements novelly, using a conservative copying algorithm that has been designed to minimise the need for explicitly copying or unsetting model elements.

Initial experiments suggest that Flock is more concise than the approaches discussed in Section 3, but we intend to more thoroughly test this hypothesis by measuring, for example, cyclomatic complexity. Other future work will involve studying the way in which Flock should be expanded to capture further concepts specific to the domain of model migration. In particular, we intend to explore the need for rule inheritance, and anticipate the addition of constructs for supporting re-use of migration strategy rules in a metamodel-independent manner.

Acknowledgement. The work in this report was supported by the European Commission via the MODELPLEX project, co-funded by the European Commission under the "Information Society Technologies" Sixth Framework Programme (2006-2009).

References

1. A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Automating co-evolution in MDE. In *Proc. EDOC*, pages 222–231. IEEE Computer Society, 2008.
2. K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646, 2006.
3. Eclipse. Generative Modelling Technologies project [online]. [Accessed 30 June 2008] Available at: `http://www.eclipse.org/gmt`, 2008.
4. Eclipse. UML2 Model Development Tools project [online]. [Accessed 7 September 2009] Available at: `http://www.eclipse.org/modeling/mdt/uml2`, 2009.
5. K. Garcés, F. Jouault, P. Cointe, and J. Bézivin. Managing model adaptation by precise detection of metamodel changes. In *Proc. ECMDA-FA*, volume 5562 of *LNCS*, pages 34–49. Springer, 2009.
6. R. Gronback. Introduction to the Eclipse Graphical Modeling Framework. In *Proc. EclipseCon*, Santa Clara, California, 2006.
7. M. Herrmannsdoerfer, S. Benz, and E. Juergens. COPE - automating coupled evolution of metamodels and models. In *Proc. ECOOP*, volume 5653 of *LNCS*, pages 52–76. Springer, 2009.
8. K. Hussey and M. 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.
9. F. Jouault and I. Kurtev. Transforming models with ATL. In *Proc. Satellite Events at MoDELS*, volume 3844 of *LNCS*, pages 128–138. Springer, 2005.
10. D.S. Kolovos. *An Extensible Platform for Specification of Integrated Languages for Model Management.* PhD thesis, University of York, United Kingdom, 2009.
11. D.S. Kolovos, R.F. Paige, and F.A.C Polack. The Epsilon Object Language (EOL). In *Proc. ECMDA-FA*, volume 4066 of *LNCS*, pages 128–142. Springer, 2006.
12. D.S. Kolovos, R.F. Paige, and F.A.C Polack. The Epsilon Transformation Language. In *Proc. ICMT*, volume 5063 of *LNCS*, pages 46–60. Springer, 2008.
13. T. Mens and S. Demeyer. *Software Evolution.* Springer-Verlag, 2007.

14. P. Muller, F. Fleurey, D. Vojtisek, Z. Drey, D. Pollet, F. Fondement, P. Studer, and J. Jzquel. On Executable Meta-Languages applied to Model Transformations. In *Model Transformations In Practice Workshop*, 2005.
15. J. Oldevik, T. Neple, R. Grønmo, J. Øyvind Aagedal, and A. Berre. Toward standardised model to text transformations. In *Proc. ECMDA-FA*, volume 3748 of *LNCS*, pages 239–253. Springer, 2005.
16. OMG. Query/View/Transformation 1.0 Specification [online]. [Accessed 23 March 2010] Available at: `http://www.omg.org/spec/QVT/1.0/`, 2008.
17. 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), 2007.
18. L.M. Rose, D.S. Kolovos, R.F. Paige, and F.A.C. Polack. An analysis of approaches to model migration. In *Proc. Joint MoDSE-MCCM Workshop*, 2009.
19. L.M. Rose, D.S. Kolovos, R.F. Paige, and F.A.C. Polack. Enhanced automation for managing model and metamodel inconsistency. In *Proc. ASE*. ACM Press, 2009.
20. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2008.
21. G. Wachsmuth. Metamodel adaptation and model co-adaptation. In *Proc. ECOOP*, volume 4609 of *LNCS*, pages 600–624. Springer, 2007.