Evolution in Model-Driven Engineering

Louis M. Rose

August 3, 2010

Contents

Chapter 6

Evaluation

Chapter 5 outlined requirements for and described the implementation of a textual modelling notation and a model migration language. This chapter describes the way in which both solutions were evaluated against their requirements, and against the thesis requirements identified in Chapter 4. A further solution, the metamodel-independent syntax, was developed in Section 5.1 and is not evaluated explicitly in this chapter, as discussed below.

The textual modelling notation (Section 5.2) was developed to address the challenges faced when performing user-driven co-evolution. Using a real-world example of user-driven co-evolution, Section 6.1 demonstrates existing and novel processes for performing manual migration, and indicates the benefits of using the textual modelling notation in the novel process.

A transformation language tailored for model migration, Epsilon Flock (Section 5.4), was developed to allow the specification and execution of model migration strategies. Flock uses a novel algorithm for relating source to target model elements, termed conservative copy. Conservative copy is evaluated against two existing mechanisms for relating source and target model elements in Section 6.2 using quantitive measurements. In addition, Epsilon Flock has been evaluated by comparison with other migration tools (Section 6.3) and with transformation languages (Section 6.4). The developers of other tools have been involved in the evaluation described in Sections 6.3 and 6.4, and their contributions are highlighted. The work presented in this chapter has been published in [Rose *et al.* 2010a, Rose *et al.* 2010d, Rose *et al.* 2010c].

Evaluation of the Metamodel-Indepenent Syntax The metamodel-independent syntax was developed to address the following thesis requirement, identified in Section 4.3: *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.*

The metamodel-independent syntax was demonstrated to facilitate loading of non-conformant models and conformance checking in Section 5.1.3. Additionally, the metamodel-independent syntax was used in the implementation of the textual modelling notation, which is evaluated in Section 6.1. As such, no further evaluation of the metamodel-independent syntax is presented in this chapter. Instead, the remainder of the chapter focuses on the evaluation of the textual modelling notation and the model migration language.

6.1 Exemplar User-Driven Co-Evolution

The analysis presented in Chapter 4 led to the discovery of user-driven co-evolution, in which model migration is not executable and is instead performed by hand. Chapter 4 highlighted two challenges faced when user-driven co-evolution techniques are applied. Firstly, model storage representations have not been optimised for use by humans, and hence user-driven co-evolution can be error-prone and time consuming. Secondly, when a multi-pass parser is used to load models (as is the case with EMF), user-driven co-evolution is an iterative process, because not all conformance errors are reported at once. These challenges led to the derivation of the following research 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 conformance report for the original model and evolved metamodel.*

Chapter 5 presented two solutions that seek to fulfil the above research requirement. The first, metamodel-independent syntax, facilitates the conformance checking of a model against any metamodel. Conformance checking can be manual (invoked by the user) or automatic (via integration of the metamodel-independent syntax with a framework for monitoring workspace changes, as described in Section 5.1.4). The second tool, an implementation of the textual modelling notation HUTN, allows models to be managed in a format that is reputedly easier for humans to use than the canonical model storage format, XMI [OMG 2004].

This section demonstrates a user-driven co-evolution process which uses the conformance reporting and textual modelling notation described in this thesis. To this end, an example of co-evolution, based on changes observed in the process-oriented project described in Chapter 4, is used throughout the remainder of this section. The example is used to show the way in which user-driven co-evolution might be achieved with and without the conformance reporting and the textual modelling notation described in Chapter 5.

6.1.1 Co-Evolution Example

The co-evolution example used throughout this section is based on changes observed in the process-oriented project, which was described in Chapter 4. The metamodel considered was developed in joint work with Adam Sampson, a research associate at the University of Kent. The work involved building a prototypical tool for editing graphical models of process-oriented programs. EuGENia [Kolovos *et al.* 2009] was used to automatically generate a graphical editor from the process-oriented metamodel. The metamodel was developed iteratively in the following manner:

1. Draw by hand a desired graphical model for a simple process-oriented program.

2. Change the metamodel to capture any new or revised domain concepts.

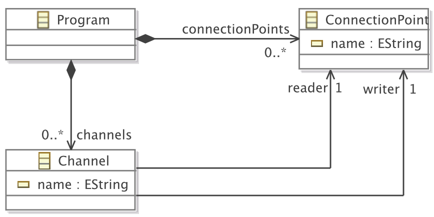
3. Regenerate the graphical editor from the metamodel.

4. Use the editor to draw the desired graphical model.

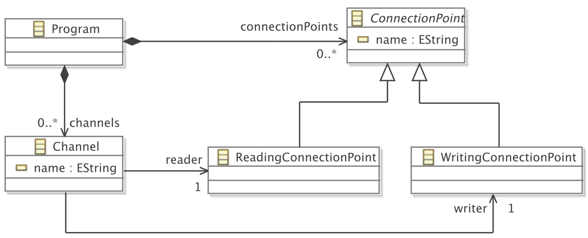
5. Check that the current (and all previous) graphical models are satisfactory representations of their hand-drawn counterparts.

After step 5, work continued by returning to step 2 if the computerised graphical model was not a satisfactory representation of the hand-drawn model created in step 1. Otherwise, work continued by returning to step 1. The metamodel was completed after 6 iterations. Each iteration produced a new pair of hand-drawn and computerised graphical models. To prevent regressions, step 5 checked the current pair and all previous pairs of models.

Here, one iteration of the process-oriented metamodel is considered, which led to the metamodel changes shown in Figure 6.1. In Figure 6.1(a), a `Program` is composed of `Channels` and `ConnectionPoints`. A `Channel` reads from and writes to exactly one `ConnectionPoint`. Further analysis of the domain revealed that `ConnectionPoints` may only be used for reading or for writing, and never both. To capture this constraint, `ConnectionPoint` was made abstract, and two subtypes, `ReadingConnectionPoint` and `WritingConnectionPoint`, were introduced. The reader and writer references of `Channel` then referred to the new subtypes, as shown in Figure 6.1(b).



(a) Original metamodel.



(b) Evolved metamodel.

Figure 6.1: Process-oriented metamodel evolution.

In general, changes made during step 2 of the process described above could cause models previously created during step 5 to become non-conformant. For the process-oriented metamodel, a user-driven co-evolution process was preferred to a developer-driven co-evolution process because they were only a small number of models (at most 5) to be migrated.

When the process-oriented metamodel was developed, no tools were available for performing user-driven co-evolution. Migration was performed by editing the storage representation of the models, which was error-prone and time-consuming. This process is now described, and is then compared to a user-driven co-evolution process that uses the conformance checking tool and the textual modelling notation described in Chapter 5.

6.1.2 Existing Process

During the development of the process-oriented metamodel, model migration was performed without user-driven co-evolution tools and involved attempting to load existing, potentially non-conformant models with EMF. Any conformance errors were noted and the corresponding XMI changed to re-establish conformance with the evolved metamodel.

For the metamodel changes shown in Figure 6.1, conformance errors were reported for all of the existing models. Initially, two types of messages were received for non-conformant models. For every instance of ConnectionPoint, the following message was produced: "Class 'ConnectionPoint' is not found or is abstract." For every instance of Channel that referenced a ConnectionPoint, the following message was produced: "Unresolved reference '<ID>' " where <ID> was the identifier of the referenced ConnectionPoint.

To fix both types of error, the XMI of each model was changed such that each instance of ConnectionPoint was replaced by an instance of ReadingConnectionPoint or WritingConnectionPoint. This involved adding an extra attribute, xsi:type, to each ConnectionPoint, which is discussed further in Section 6.1.4.

In a small number of cases, the wrong subtype of ConnectionPoint was selected, probably because XMI identifies elements using randomly generated strings. In one model, two connection points named a_reader and a_writer had the very similar XMI IDs _MeFREC8sEd69s-McmXQlqQ and _M7EvEC8sEd69s-McmXQlqQ, respectively. Because of this, the two connection points were assigned the wrong types when the XMI was changed by hand.

Conformance errors are reported only when a model is loaded by EMF (and hence, in this case, only when the graphical editor is used to open a model). In other words, when the XMI of a model is changed by hand, conformance is not checked when the model is saved to disk. When the wrong subtype of ConnectionPoint was selected, the following message was produced when the model was opened with the graphical editor: "Value 'po.impl.ReadingConnectionPoint@7fde1684 (name: a_writer)' is not legal." After changing xsi:type attributes to instantiate the correct subtypes of ConnectionPoint, all of the models could be opened without error by the graphical editor.

6.1.3 Proposed Process

A user-driven co-evolution process using the conformance reporting tool and the textual modelling notation, HUTN, presented in Chapter 5 is now described. The new process involves invoking the conformance reporting tool to determine which models have conformance problems, generating HUTN for each non-conformant model, and fixing the conformance problems in the generated HUTN.

For the metamodel changes shown in Figure 6.1, the conformance reporting tool reports three types of error message when invoked on non-conformant models. For every instance of ConnectionPoint, the following message is produced: "Cannot instantiate the abstract class: ConnectionPoint." For every instance of `Channel`, the following two error messages are produced: "Expected ReadingConnectionPoint for: reader" and "Expected WritingConnectionPoint for: writer."

To fix the errors, a HUTN representation of each non-conformant model is generated by invoking the "Generate HUTN" context menu item. Figure 6.2 shows the HUTN generated for one of the non-conformant process-oriented models. Fixing the conformance problems involves changing the HUTN source by hand (and then regenerating the XMI using the "Generate Model" context menu item). For the model shown in Figure 6.2, fixing the conformance problems involved changing the type of `a_reader` to `ReadingConnectionPoint` and the type of `a_writer` to `WritingConnectionPoint`. Whenever the user saves the HUTN document, both syntax and conformance are checked by the background incremental compiler. Any problems are reported while the model is migrated.

6.1.4 Summary

This section has demonstrated existing and new user-driven co-evolution processes using an example taken from a real-world project in which a metamodel was developed incrementally. Comparison of the two processes highlights several benefits of the proposed process, which used tools described in Chapter 5.

Firstly, the conformance reporting tool presented in Section 5.1 can report more types of conformance problem at once than the model loading mechanism of EMF because the former uses a multi-pass parser, while the latter uses a single-pass parser. For the metamodel changes shown in Figure 6.1(b), both the conformance reporting tool and the EMF loading mechanism reported that the evolved metamodel did not permit instantiation of the abstract class `ConnectionPoint`. Another type of conformance problem – the type of `Channel#reader` (`Channel#writer`) must be a `ReadingConnectionPoint` (`WritingConnectionPoint`) – was reported by the conformance reporting tool when it was first invoked and reported by the loading mechanism of EMF only when it was invoked after the first category of conformance problem was fixed.

Secondly, the implementation of HUTN described in Section 5.2 uses a background incremental compiler that checks both the syntax and conformance of the HUTN source code. On the other hand, EMF checks conformance only when a model is loaded (by the graphical model editor, in this case). Saving XMI to disk does not cause EMF to check its conformance.

Thirdly, migration involved changing the types of some model elements. In XMI, when the type of a model element can be inferred from the context in which it is instantiated, type information is omitted. This reduces the size of the model on disk, but can be problematic for model migration. For example, in the original process-oriented metamodel (Figure 6.1(a)) any model element contained in the `Program#connectionPoints` reference must be an instance of `ConnectionPoint`, and type information can be omitted from the XMI. When the process-oriented metamodel evolved to allow `ReadingConnectionPoint` and `WritingConnectionPoints` to be contained in the `Program#connectionPoints`, type information must be added. To add type information to XMI, the person performing migration must know the correct syntax, for example: `xsi:type="po.ReadingConnectionPoint"`. By contrast, the

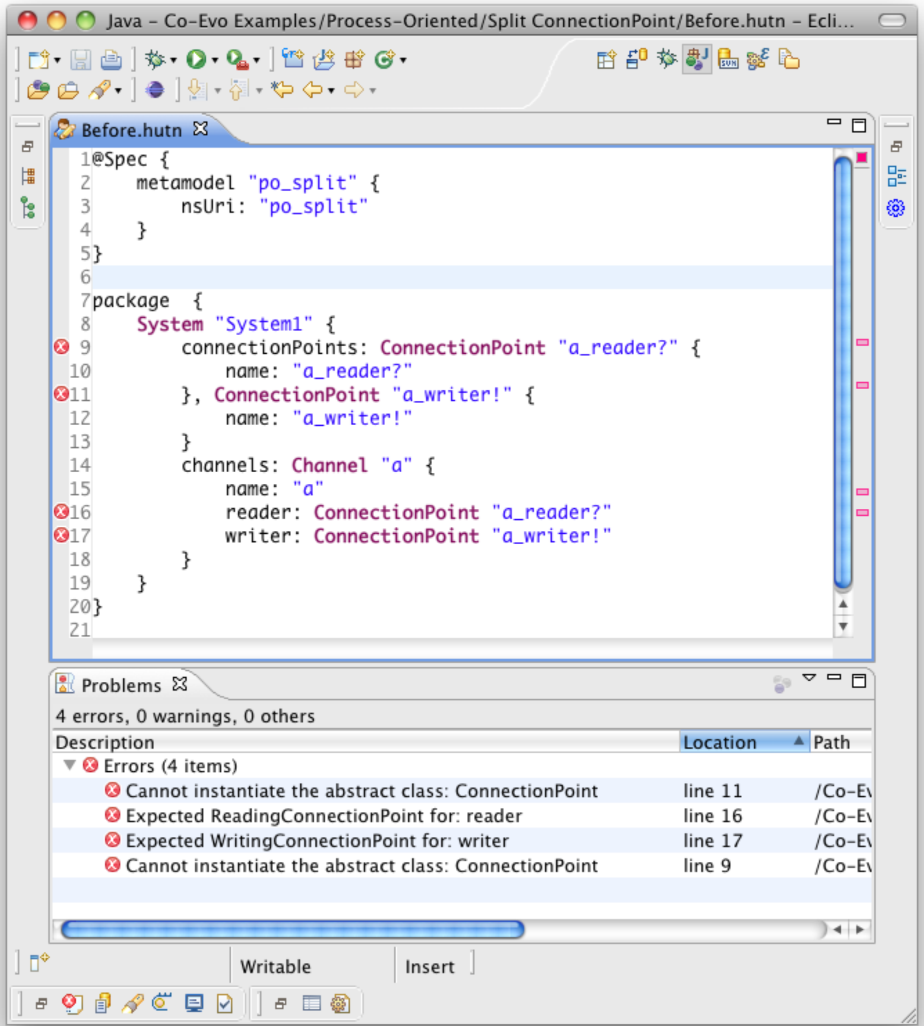


Figure 6.2: HUTN for a non-conformant process-oriented model.

type of every model element is declared explicitly in HUTN. As such, every HUTN document contains examples of how type information should be specified. Hence, changing the type of a model element in HUTN is arguably more straightforward than in XMI.

Finally, migration involved understanding which `Channels` referenced which `ConnectionPoints`. By default, EMF uses universally unique identifiers (UUIDs) such as `_M7EvEC8sEd69s-McmXQlqQ` – or URI fragments (document-specific relative paths) such as `@connectionPoints.0` – to identify model elements. By contrast, the implementation of HUTN described in Chapter 5, uses the value of a model element's name feature (where one is defined) to identify model elements. For example, in Figure 6.2 the `Channel` on line 14 refers to `ConnectionPoints` by name (lines 16 and 17). Hence, referencing model elements in HUTN is arguably more straightforward than in XMI.

Further research is required to more rigorously assess the differences between the two user-driven co-evolution processes discussed in this section. In particular, the textual modelling notation used in the proposed process, HUTN, purports to be human-usable [OMG 2004], but no usability studies have compared HUTN with other model representations, such as XMI. The implementation of tools for performing user-driven co-evolution, described in Chapter 5, enable further comparisons, but a thorough investigation of their usability is beyond the scope of this thesis.

This section has used two of the tools described in Chapter 5 to demonstrate a user-driven co-evolution process that provides a conformance report and allows the editing of non-conformant models in a textual modelling notation. The benefits of the proposed process have been highlighted by comparison to an existing user-driven co-evolution process using a co-evolution example, taken from a project that used user-driven co-evolution.

6.2 Quantitive Comparison of Model Migration Languages

In Section 4.3, the following research requirement was identified: *This thesis must implement and evaluate a domain-specific language for specifying and executing model migration strategies, comparing it to existing languages for specifying model migration strategies.* As discussed in Section 5.4, this thesis contributes Epsilon Flock, a domain-specific language for model migration. This section approches the second part of the above research requirement by comparing Flock with languages that are used in contemporary migration tools.

In developer-driven migration, a programming language codifies the migration strategy. Because migration involves deriving the migrated model from the original, migration strategies typically access information from the original model and, based on that information, update the migrated model in some way. As such, migration is written in a language with constructs for accessing and updating the original and migrated models. Here, those language constructs are termed *model operations*. Using examples of co-evolution, this section explores the variation in frequency of model operation over different model migration languages, and discusses to what extent the results of this comparison can be used to assess the suitability of the languages considered for model migration.

As discussed in Chapter 5, the languages currently used for model migration vary. Model-to-model transformation languages are used in some migration tools (e.g. [Cicchetti *et al.* 2008, Garcés *et al.* 2009]), and general-purpose languages in others (e.g. [Herrmannsdoerfer *et al.* 2009b, Hussey & Paternostro 2006]). Irrespective of the language used for migration, the way in which a migration tool relates original and migrated model elements falls into one of two categories: new- or existing-target, which were first introduced in Section 5.3.2. In the former, the migrated model is created afresh by the execution of the migration strategy. In the latter, the migrated model is initialised as a copy of the original model and then the migration strategy is executed.

Flock contributes a novel approach for relating original and migrated model elements, termed conservative copy. Conservative copy is a hybrid of new- and existing-target approaches. This section compares new-target, existing-target and conservative copy in the context of model migration. Section 6.2.1 describes the data used in the comparison. The method for the comparison is discussed in Section 6.2.2. Section 6.2.3 identifies model operations for each of the migration languages used in the comparison, and Section 6.2.4 presents and analysis the results.

6.2.1 Data

Five examples of co-evolution were used to compare new-target, existing-target and conservative copy. This section briefly discusses the data used in the comparison.

Co-evolution Examples

To remove one of the possible threats to the validity of the comparison, the examples used were distinct from those used to identify the thesis requirements in Chapter 4. The five examples used in this section are taken from three projects.

Two examples were taken from the *Newsgroup* project, which performs statistical analysis of NNTP newsgroups and was developed by Dimitris Kolovos, a lecturer in this department. One example was taken from *UML* (the Unified Modeling Language), an OMG specification of a language for modelling software systems. Two examples were taken from *GMF* (Graphical Modeling Framework) [Gronback 2009], an Eclipse project for generating graphical model editors.

Selection of Migration Languages

As discussed above, there are two ways in which existing migration languages relate original and migrated model elements, new- and existing-target. Flock contributes a third way, conservative copy. For the comparison with Flock, one new- and one existing-target language was chosen.

The Atlas Transformation Language (ATL), a model-to-model transformation language that has been used in [Cicchetti *et al.* 2008, Garcés *et al.* 2009] for model migration. As discussed in Section 5.3.2, model-to-model transformation languages support only new-target transformations for model migration[1].

The author is aware of two approaches to migration that use existing-target transformations. In COPE [Herrmannsdoerfer *et al.* 2009b], migration strategies can be hand-written in Groovy when no co-evolutionary operator is applicable. As discussed in Section 5.3.2, COPE's Groovy migration strategies use an existing-target approach. COPE provides six functions for interacting with model elements, such as `set`, for changing the value of a feature, and `unset`, for removing all values from a feature. In the remainder of this section, the term *Groovy-for-COPE* is used to refer to the combination of the Groovy programming language and the functions provided by COPE for use in hand-written migration strategies. In Ecore2Ecore [Hussey & Paternostro 2006], migration is performed when the original model is loaded, effectively an existing-target approach.

The comparison to Flock described in this section uses ATL to represent new-target approaches and Groovy-for-COPE to represent existing-target approaches. Groovy-for-COPE was preferred to Ecore2Ecore because the latter is not as expressive[2] and cannot be used for migration in the co-evolution examples considered here.

6.2.2 Method

For each example of co-evolution, a migration strategy was written using each migration language (ATL, Groovy-for-COPE and Flock). The co-evolution examples considered included reference migrated models. The correctness of each migration strategy was assured by comparing the migrated model produced with the corresponding reference migrated model.

Next, a set of model operations were identified, as described in Section 6.2.3 and a program was written to count the frequency of each model operation in each migration strategy. The counting program was tested by comparison to manual counts.

The method described above introduces one non-trivial threat to the validity of the comparison. Because the author is obviously more familiar with Flock than with ATL and Groovy-for-COPE, it is possible that the migration strategies written in the latter two languages may contain more model operations than necessary. In some cases, it was possible to reduce the effects of this threat by re-using or adapting existing migration strategy code written by the migration language authors. This is discussed further in the sequel.

6.2.3 Model Operations

The way in which model operations were counted is now described. Two categories of model operation were considered: copying and deleting. Copying (deleting) operations are used to assign (remove) values to (from) elements of the migrated model. The three languages considered use different syntax for semantically equivalent statements. Some of the languages provide more than one construct for copying or deleting values. The remainder of this section lists

[1]Because, in model migration, the source and target metamodels are not the same.

[2]Communication with Ed Merks, Eclipse Modeling Project leader, 2009, available at `http://www.eclipse.org/forums/index.php?t=tree&goto=486690&S=b1fdb2853760c9ce6b6b48d3a01b9aac`

the concrete syntax of the model operations in each language. In addition, the extent to which the comparison described in this section was able to use code written by the authors of each language is discussed.

Atlas Transformation Language (ATL)

For the Atlas Transformation Language (ATL), the following model operations were counted:

- **Copying:** Assignment to a feature:

  ```
  <feature> <- <value>
  ```

Deleting operations are not used in new-target migration strategies. A new-target migration strategy specifies only those values that must appear in the migrated model and, unlike existing-target approaches and conservative copy, no values are copied automatically prior to the execution of the migration.

Groovy-for-COPE

For Groovy-for-COPE, the following model operations were counted:

- **Copying:** Assignment to a feature:

  ```
  <element>.<feature> = <value>
  <element>.<feature>.add(<value>)
  <element>.<feature>.addAll(<collection_of_values>)
  <element>.set(<feature>) = <value>
  ```

- **Deleting:** Unsetting a feature:

  ```
  <element>.<feature>.unset()
  ```

- **Deleting:** Removing a model element:

  ```
  delete <element>
  ```

Groovy-for-COPE provides four assignment statements. The first is typical. The second and third are used to add values to a collection. The fourth is used for reflective access to a model.

The deleting operations are necessary for some existing-target migration strategies, because the migrated model (which is initialised as a copy of the original model) may contain data that is no longer captured in the evolved metamodel.

As discussed in Section 4.2.3, COPE provides a library of built-in, reusable co-evolutionary operators. Each co-evolutionary operator specifies a metamodel evolution along with a corresponding model migration strategy. To mitigate the chance that the author, who is less familiar with COPE than Flock, introduced additional model operations, writing the Groovy-for-COPE migration strategies involved, where possible, applying an appropriate COPE co-evolutionary operator and counting the number of model operations in the generated migration strategy. Not all of the examples considered could be specified using co-evolutionary operators, and, in these cases, the Groovy-for-COPE migration strategy was written entirely by the author.

	Migration Language Source-Target Relationship		
(Project) Example	**ATL** New	**G-f-C** Existing	**Flock** Conservative
(Newsgroup) Extract Person	9	6	5
(Newsgroup) Resolve Replies	8	3	2
(UML) Activity Diagrams	15	15	8
(GMF) Graph	101	11	14
(GMF) Gen2009	310	16	16

Table 6.1: Model operation frequency (evaluation examples).

Epsilon Flock

Epsilon Flock, a transformation language tailored for model migration, was developed in this thesis and discussed in Chapter 5. Flock uses the Epsilon Object Language (EOL) [Kolovos *et al.* 2006c] to access and update model values. For Flock, the following model operations were counted:

- **Copying:** Assignment to a feature:

```
<element>.<feature> := <value>
<element>.<feature>.add(<value>)
<element>.<feature>.addAll(<collection_of_values>)
```

- **Deleting:** Removing a model element:

```
delete <element>
```

Like Groovy-for-COPE, Flock provides several assignment statements. The first is typical. The second and third are used to add values to a collection.

Flock provides a remove operation but not an unset. The former is required to remove model elements that no longer conform to the target metamodel. Flock does not provide an unset operation because the conservative copy algorithm will never copy to the migrated model any value that does not conform to the evolved metamodel.

6.2.4 Results

By measuring the number of model operations in model migration strategies, the way in which each co-evolution approach relates original and migrated model elements was investigated. The five examples of model migration discussed above were measured to obtain the results shown in Table 6.1.

In addition, the results from measuring the examples identified from Chapter 4 are shown in Table 6.2. However, because the examples used to produce the measurements shown in Table 6.2 were used to design Flock, the measurements in Table 6.2 are less relevant to the evaluation presented here than the measurements shown in Table 6.1. Nevertheless, the measurements made in Table 6.2 are included in the interest of transparency.

For four of the five examples in Table 6.1, conservative copy requires less model operations than new-target and existing-target. The GMF Graph example is the exception to this trend.

	Migration Language Source-Target Relationship		
(Project) Example	**ATL** New	**G-f-C** Existing	**Flock** Conservative
(FPTC) Connections	6	6	3
(FPTC) Fault Sets	7	5	3
(GADIN) Enum to Classes	4	1	0
(GADIN) Partition Cont	5	3	2
(Literature) PetriNets	12	10	6
(Newsgroup) Extract Person	9	6	5
(Newsgroup) Resolve Replies	8	3	2
(Process-Oriented) Split CP	8	1	1
(Refactor) Cont to Ref	4	5	3
(Refactor) Ref to Cont	3	4	3
(Refactor) Extract Class	5	4	2
(Refactor) Extract Subclass	6	0	0
(Refactor) Inline Class	4	5	2
(Refactor) Move Feature	6	2	1
(Refactor) Push Down Feature	6	0	0

Table 6.2: Model operation frequency (analysis examples).

For all of the five examples in Table 6.1, no migration strategy specified with existing-target contained less model operations when encoded with new-target. However, three of the Refactor examples in Table 6.2 do not follow this trend.

The results are now investigated, starting by discussing the differences between the source-target relationships. Investigating the results led to the discovery of two limitations of the conservative copy implementation in Flock, relating to sub-typing and side-effects during initialisation. These limitations are also discussed below.

Source-Target Relationships

New-target, existing-target and conservative copy initialise the migrated model in a different way. New-target initialises an empty model, while existing-target initialises a complete copy of the original model. Conservative copy initialises the migrated model by copying only those model elements from the original model that conform to the migrated metamodel.

New- and existing-target are opposites. In the former, explicit assignment operations must be used to copy values from original to migrated model for each feature that is not affected by the metamodel evolution. By contrast, in the latter unset operations must be used when the value of a feature should not have been copied.

In situations where a large number of metamodel features have not been affected by evolution, expressing migration with a new-target transformation language requires more model operations than using an existing-target transformation language. This is particularly noticeable in the GMF examples shown in Table 6.1, where ATL (new-target) requires many more model operations than Groovy-for-COPE (existing-target) and Flock (conservative copy).

In situations where a large number of metamodel features have been renamed, expressing migration with an existing-target transformation language requires more model operations than using a new-target transformation language. This is because, in an existing-target transformation language, two model operations (an unset and an assignment) are needed to migrate values in response to the renaming of a feature:

```
<element>.<newFeature> = <element>.unset(<oldFeature>)
```

By contrast, a new-target transformation language requires only one model operation (an assignment):

```
<migrated_element>.<feature> = <original_element>.<feature>
```

The UML (Table 6.1) and Refactor Inline Class (Table 6.2) examples contained several feature renamings, and consequently the Groovy-for-COPE (existing-target) figure was nearer to the ATL (new-target) figure than the Flock (conservative copy) figure. This is contrary to the trend in Tables 6.1 and 6.2.

Conservative copy is a hybrid of new- and existing target. Model values that have been affected by evolution are not copied to the migrated model, and so the migration strategy need not unset affected model values. Model values that have not been affected by evolution are copied to the migrated model, and so the migration strategy need not explicitly copy unaffected model values.

Two conclusions can be drawn from this discussion. Firstly, in general, less model operations are used when specifying a migration strategy with a conservative copy migration language than when specifying the same migration strategy with a new- or existing-target migration language.

Secondly, in the examples studied here, there are often more features unaffected by metamodel evolution than affected. Consequently, specifying model migration with a new-target migration language requires more model operations than in an existing-target migration language for the examples shown in Tables 6.1 and 6.2. [Sprinkle 2003] proposes that metamodel evolution often involves changes to relatively few metamodel elements, and the results presented in this section support his argument.

Subtyping

Form the GMF Graph example (Table 6.1), conservative copy requires more model operations than existing-target. Investigation of this result revealed a limitation in conservative copy limitation in Flock, which is now described.

Figure 6.3 shows a simplified part of the GMF Graph metamodel prior to evolution. When the metamodel evolved, the types of the `figure` and `accessor` features were changed. Consequently, the migration strategy needed to change the values stored in the `figure` and `accessor` features. In the simplified example presented here, the types of the `figure` and `accessor` features were changed from string to integer. The intended migration semantics are for the integer value to be the length of the original string value. This is a simplification that is representative of the actual GMF Graph metamodel evolution.

In ATL, the migration strategy for the metamodel evolution discussed above can be expressed using two model operations, because an ATL transformation rule may inherit the body of another. The `DiagramElements` rule on lines 1-4 of Listing 6.1 specifies that the value of the figure feature should be the length of the original value. For `Nodes`, `Connections` and `Compartments`, migration can be specified simply by extending the `DiagramElements` rule (lines 6-19).

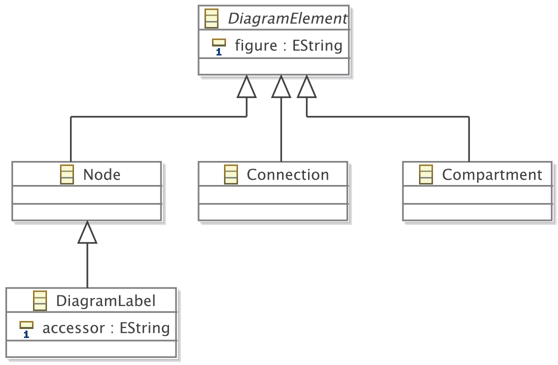


Figure 6.3: Simplified fragment of the GMF Graph metamodel.

For `DiagramLabels`, the values of both the accessor and figure feature must be migrated. On lines 21-24, the `DiagramLabels` extends `Nodes` and hence `DiagramElements` to inherit the body of the latter for migrating figures, and, in addition, the `DiagramLabels` rule defines the migration for the value of the `accessor` feature.

```
1  abstract rule DiagramElements {
2    from o : Before!DiagramElement
3    to  m : After!DiagramElement ( figure <- o.figure.length() )
4  }
5
6  rule Nodes extends DiagramElements {
7    from o : Before!Node
8    to  m : After!Node
9  }
10
11 rule Connections extends DiagramElements {
12   from o : Before!Connection
13   to  m : After!Connection
14 }
15
16 rule Compartments extends DiagramElements {
17   from o : Before!Compartment
18   to  m : After!Compartment
19 }
20
21 rule DiagramLabels extends Nodes {
22   from o : Before!DiagramLabel
23   to  m : After!DiagramLabel ( accessor <- o.accessor.length() )
24 }
```

Listing 6.1: Simplified GMF Graph model migration in ATL

In Groovy-for-COPE, the migration is similar to ATL. However, Groovy-for-COPE is entirely imperative, and so the migration (Listing 6.2) is more concise than the ATL migration (Listing 6.1). In Listing 6.2, a loop iterates over each instance of `DiagramElement` (line 1), migrating the value of its figure feature (line 2). If the `DiagramElement` is also a `DiagramLabel` (line 4), the value of its accessor feature is also migrated (line 5).

```
1  for (diagramElement in subtyping.DiagramElement.allInstances()) {
2    diagramElement.figure = diagramElement.figure.length()
```

```

  if (subtyping.DiagramLabel.allInstances.contains(diagramElement)) {
   diagramElement.accessor = diagramElement.accessor.length()
  }
}
```

Listing 6.2: Simplified GMF Graph model migration in COPE

In both ATL and Groovy-for-COPE, only 2 model operations are required for this migration: an assignment for each of the two features being migrated. However, the equivalent Flock migration strategy, shown in Listing 6.3, requires 5 model operations. In Flock, a migrate rule must be specified for each concrete subtype of `DiagramElement`. A single `DiagramElement` rule cannot be used to migrate the concrete subtypes because, when a rule does not specify a `to` part, Flock will create an instance of the type named after the `migrate` keyword (i.e. `DiagramElement` here). Because `DiagramElement` is abstract, Flock will fail with a runtime error. Furthermore, because only one rule can be applied to each original model element, the `DiagramLabel` rule (lines 9-12) must migrate the values of both the figure and accessor features, and cannot exploit the kind of re-use provided by ATL with rule inheritance.

```
1  migrate Compartment {
2   migrated.figure := original.figure.length();
3  }
4
5  migrate Connection {
6   migrated.figure := original.figure.length();
7  }
8
9  migrate DiagramLabel {
10  migrated.figure := original.figure.length();
11  migrated.accessor := original.accessor.length();
12 }
13
14 migrate Node {
15  migrated.figure := original.figure.length();
16 }
```

Listing 6.3: Simplified GMF Graph model migration in Flock

The example presented in this section highlights a limitation of the conservative copy algorithm as it is implemented in Flock. The extent to which this limitation can be addressed with changes to the implementation of Flock, and in general, is discussed in Section 7.2. This section now considers one further limitation of existing-target and conservative copy migration languages.

Side-Effects during Initialisation

The measurements observed for one of the examples of co-evolution from Chapter 4, Change Reference to Containment, cannot be explained by the conceptual differences between source-target relationship. Instead, the way in which the source-target relationship is implemented must be considered.

When a reference feature is changed to a containment reference during metamodel evolution, constructing the migrated model by starting from the original model (as is the case with existing-target and conservative copy) can have side-effects which complicate migration.

In the Change Reference to Containment example, a `System` initially comprises `Ports` and `Signatures` (Figure 6.4). A `Signature` references any number of `ports`. The metamodel is to be evolved so that `Ports` can no longer be shared between `Signatures`.

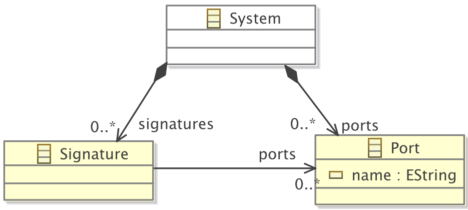


Figure 6.4: Original metamodel.

The evolved metamodel is shown in Figure 6.5. `Signatures` now contain - rather than reference - `Ports`. Consequently, the `ports` feature of `System` is no longer required and is removed.

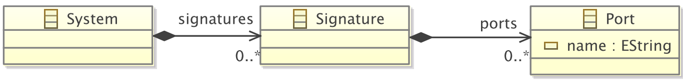


Figure 6.5: Evolved metamodel.

The migration strategy is straightforward in a new-target migration language: for each `Signature` in the original model, each member of the `ports` feature is cloned and added to the `ports` feature of the equivalent `Signature`.

```
rule Systems {
  from
    o : Before!System
  to
    m : After!System ( signatures <- o.signatures )
}

rule Signature {
  from
    o : Before!Signature
  to
    m : After!Signature (
      ports <- o.ports->collect(p | thisModule.Port(p))
    )
}

lazy rule Port {
  from
    o : Before!Port
  to
    m : After!Port ( name <- o.name )
```

Listing 6.4: Change R to C model migration in ATL

In existing-target and conservative copy migration languages, migration is less straightforward because the value of a containment reference (`Signature#ports`) is set automatically by the migration strategy execution engine. When a containment reference is set, the contained objects are removed from their previous containment reference (i.e. setting a containment reference can have side-effects). Therefore, in a `System` where more than one `Signature` references the same `Port`, the migrated model cannot be formed by copying the contents of `Signature#ports` from the original model. Attempting to do so causes each `Port` to be contained only in the last referencing `Signature` that was copied.

In COPE, conformance is checked only after execution of the migration strategy, when the model is transformed to a metamodel-specific representation. Therefore, the containment nature of the reference is not enforced until after the migration strategy is executed. Hence, the migration strategy discussed here can be specified by unsetting the contents of the `ports` reference (line 4 of Listing 6.5), and creating a copy of each referenced `Port` (lines 5-7 of Listing 6.5).

Unlike the ATL migration strategy, the ports in the Groovy-for-COPE migration strategy are cloned in the same model as the original port. Consequently, the Groovy-for-COPE migration strategy must either only clone ports that are referenced by more than one signature or clone every referenced port, but delete all of the original ports. The latter approach requires 2 more model operations (to populate and delete the original ports) than the former (shown in Listing 6.5).

```
1  def contained = []
2
3  for(signature in refactorings_changeRefToCont.Signature.
       allInstances) {
4    for(port in signature.ports)) {
5      // when more than one Signature references this port
6      if (contained.contains(port)) {
7        def clone = Port.newInstance()
8        clone.name = port.name
9        signature.ports.add(clone)
10       signature.ports.remove(port)
11     } else {
12       contained.add(port)
13     }
14   }
15 }
16
17 for(port in refactorings_changeRefToCont.Port.allInstances) {
18   if (not refactorings_changeRefToCont.Signature.allInstances.any {
         it.ports.contains(port) }) {
19       port.delete()
20   }
21 }
```

Listing 6.5: Change R to C model migration in COPE

In Flock, the containment nature of the reference is enforced when the migrated model is initialised. Because changing the contents of a containment reference can have side-effects, a `Port` that appears in the `ports` reference of a `Signature` in the original model may not have been automatically copied to the `ports` reference of the equivalent `Signature` in the migrated model during initialisation. Consequently, the migration strategy must check the `ports` reference of each migrated `Signature`, cloning only those `Ports` that have not be automatically copied during initialisation (see line 3 of Listing 6.6).

```
1  migrate Signature {
2    for (port in original.ports) {
3      if (migrated.ports.excludes(port.equivalent())) {
4        var clone := new Migrated!Port;
5        clone.name := port.name;
6        migrated.ports.add(clone);
7      }
8    }
9  }
10
11 delete Port when: not Original!Signature.all.exists(s|s.ports.
       includes(original))
```

Listing 6.6: Change R to C model migration in Flock

The Groovy-for-COPE and Flock migration strategies must also remove any `Ports` which are not referenced by any `Signature` (lines 17-21 of Listing 6.5, and line 11 of Listing 6.6 respectively), whereas the ATL migration strategy, which initialises any empty migrated model, does not copy unreferenced `Ports`.

When a non-containment reference is changed to a containment reference, producing a migration strategy in Flock and Groovy-for-COPE requires the user to be aware of the side-effects that can occur during initialisation. It may be possible to extend the existing-target and conservative copy algorithms used in COPE and Flock to automatically perform cloning when a reference is changed to be a containment reference. This is discussed further in Section 7.2.

6.2.5 Summary

By measuring frequency of model operations, this section has compared, in the context of model migration, three approaches to relating source-target relationship: new-target, existing-target and conservative copy. The results have been analysed and the measurement method described.

The analysis of the measurements has shown that new- and existing-target migration languages are better suited to different contexts. New-target languages require less model operations than existing-target languages when metamodel evolution involves the renaming of features. Existing-target languages require less model operations than new-target languages when metamodel evolution does not affect most model elements. For the examples considered here, the latter context was more common. Conservative copy requires less model operations than both new- and existing-target in almost all of the examples considered here.

The comparison has highlighted two limitations of the conservative copy algorithm implemented in Epsilon Flock, and this section has shown how these limitations are problematic for specifying some types of migration strategy.

The author is not aware of any existing quantitive comparisons of migration languages, and, as such, the best practices for conducting such comparisons are not clear. The method used in obtaining these measurements has been described to provide a foundation for future comparisons.

6.3 Migration Tool Comparison

As discussed in Chapter 3, several tools for managing co-evolution are described in the literature. Chapter 5 proposes a further tool, Epsilon Flock. While each tool has strengths and weaknesses, little is known about how migration tools compare in practice, which makes tool selection more challenging. Consequently. Chapter 4 identified the following thesis requirements: *This thesis must compare and evaluate existing languages for specifying model migration strategies. This thesis must implement and evaluate a domain-specific language for specifying and executing model migration strategies,* ***comparing it to existing languages for specifying model migration strategies****.*

To approach these two requirements, this section describes work that compares Flock and three existing model migration that were selected from those described in Chapter 4. Following the process outlined in Section 6.3.1, the tools were applied to two co-evolution examples to facilitate their comparison. To improve the validity of the comparison, the developers of each tool were invited to participate. The remainder of this section reports our experiences with each tool (Section 6.3.2), and synthesises advice and guidelines for identifying the most appropriate model migration tool in different situations (Section 6.3.3).

This section is based on joint work with Markus Herrmannsdöerfer (a research student at Technische Universität München), James Williams (a research student in this department), Dimitrios Kolovos (a lecturer in this department) and Kelly Garcés (a research student at EMN-INRIA / LINA-INRIA in Nantes), and has been published in [Rose *et al.* 2010a]. Garcés provided assistance with installing and configuration one of the migration tools, and commented on a draft of the paper. Herrmannsdöerfer, Williams and Kolovos played a larger role in the comparison. Here, the work is narrated to make clear their contributions.

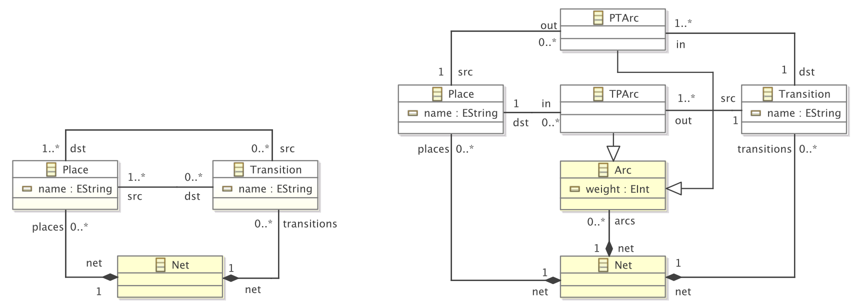
6.3.1 Comparison Method

The comparison described in this section is based on practical application of the tools to the co-evolution examples described below. This section also discusses the tool selection and comparison processes. Herrmannsdöerfer and I identified the co-evolution examples, and formulated the comparison process.

Co-Evolution Examples

To compare migration tools, two examples of co-evolution were used. The first is a well-known problem in the model migration literature and was used to test the installation and configuration of the migration tools, as discussed in Section 6.3.1. The second is a larger example taken from a real-world model-driven development project, and was identified as a potentially useful example for co-evolution case studies in Chapter 4 and in [Herrmannsdoerfer *et al.* 2009a].

Petri Nets. The first example is an evolution of a Petri net metamodel, previously used to describe the implementation of Epsilon Flock (Section 5.4), and in [Cicchetti *et al.* 2008, Garcés *et al.* 2009, Wachsmuth 2007] to discuss co-evolution and model migration.



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

Figure 6.6: Petri nets metamodel evolution (taken from [Rose *et al.* 2010e]). Shading is irrelevant.

In Figure 6.6(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 6.6(a) is to be evolved to support weighted connections between `Places` and `Transitions` and between `Transitions` and `Places`.

The evolved metamodel is shown in Figure 6.6(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`.

GMF. The second example is taken from the Graphical Modeling Framework (GMF) [Gronback 2009], an Eclipse project for generating graphical editors for models. The development of GMF is model-driven and utilises four domain-specific metamodels. Here, we consider one of those metamodels, GMF Graph, and its evolution between GMF versions 1.0 and 2.0.

The GMF Graph metamodel (not illustrated) describes the appearance of the generated graphical model editor. The metaclasses `Canvas`, `Figure`, `Node`, `DiagramLabel`, `Connection`, and `Compartment` are used to represent components of the graphical model editor to be generated. The evolution in the GMF Graph metamodel was driven by analysing the usage of the `Figure#referencingElements` reference, which relates `Figures` to the `DiagramElements` that use them. As described in the GMF Graph documentation[3], the `referencingElements` reference increased the effort required to re-use figures, a common activity for users of GMF. Furthermore, `referencingElements` was used only by the GMF code generator to determine whether an accessor should be generated for nested `Figures`.

In GMF 2.0, the Graph metamodel was evolved to make re-using figures more straightforward by introducing a proxy [Gamma *et al.* 1995] for `Figure`, termed `FigureDescriptor`. The original `referencingElements` reference was removed, and an extra metaclass, `ChildAccess`, was added to make more explicit the original purpose of `referencingElements` (accessing nested `Figures`).

[3]`http://wiki.eclipse.org/GMFGraph_Hints`

GMF provides a migrating algorithm that produces a model conforming to the evolved Graph metamodel from a model conforming to the original Graph metamodel. In GMF, migration is implemented using Java. The GMF source code includes two example editors, for which the source code management system contains versions conforming to GMF 1.0 and GMF 2.0. For the comparison of migration tools described in this paper, the migrating algorithm and example editors provided by GMF were used to determine the correctness of the migration strategies produced by using each model migration tool.

Compared Tools

The comparison described in this section included one tool from each of the three categories identified in Chapter 4 – *manual specification*, *operator-based* and *metamodel matching* approaches. The tools selected were Epsilon Flock, COPE [Herrmannsdoerfer *et al.* 2009b] and the AtlanMod Matching Language (AML) [Garcés *et al.* 2009], respectively. A further tool from the manual specification category, Ecore2Ecore, was included because it is distributed with the Eclipse Modeling Framework (EMF), arguably the most widely used modelling framework. AML, COPE and Ecore2Ecore were discussed in Chapter 4, and Epsilon Flock in Chapter 5.

Comparison Process

The comparison of migration tools was conducted by applying each of the four tools (Ecore2Ecore, AML, COPE and Flock) to the two examples of co-evolution (Petri nets and GMF). The developers of each tool were invited to participate in the comparison. The authors of COPE and Flock were able to participate fully, while the authors of Ecore2Ecore and AML were available for guidance, advice, and to comment on preliminary results.

Each tool developer was assigned a migration tool to apply to the two co-evolution examples. Because the authors of Ecore2Ecore and AML were not able to participate fully in the comparison, two colleagues experienced in model transformation and migration, James Williams and Dimitrios Kolovos, stood in. To improve the validity of the comparison, each tool was used by someone other than its developer. Other than this restriction, the tools were allocated arbitrarily.

The comparison was conducted in three phases. In the first phase, criteria against which the tools would be compared were identified by discussion between the tool developers. In the second phase, the first example of co-evolution (Petri nets) was used for familiarisation with the migration tools and to assess the suitability of the comparison criteria. In the third phase, the tools were applied to the larger example of co-evolution (GMF) and results were drawn from the experiences of the tool developers. Table 6.3 summarises the comparison criteria used, which provide a foundation for future comparisons. The next section presents, for each criterion, observations from applying the migration tools to the co-evolution examples.

Table 6.3: Summary of comparison criteria.

Name	Description
Construction	Ways in which tool supports the development of migration strategies
Change	Ways in which tool supports change to migration strategies
Extensibility	Extent to which user-defined extensions are supported
Re-use	Mechanisms for re-using migration patterns and logic
Conciseness	Size of migration strategies produced with tool
Clarity	Understandability of migration strategies produced with tool
Expressiveness	Extent to which migration problems can be codified with tool
Interoperability	Technical dependencies and procedural assumptions of tool
Performance	Time taken to execute migration

6.3.2 Comparison Results

This section reports the similarities and differences of each tool, using the nine criteria described above. The migration strategies formulated with each tool are available online[4].

Each subsection below considers one criterion. This section reports the experiences of the developer to which each tool was allocated. As such, this section contains the work of others. Specifically, Herrmannsdöerfer wrote about Epsilon Flock, Williams wrote about COPE and Kolovos wrote about Ecore2Ecore. (I wrote about AML, and the introductions to each criterion).

Constructing the migration strategy

Facilitating the specification and execution of migration strategies is the primary function of model migration tools. This section reports the process for and challenges faced in constructing migration strategies with each tool.

AML. An AML user specifies a combination of match heuristics from which AML infers a migrating transformation by comparing original and evolved metamodels. Matching strategies are written in a textual syntax, which AML compiles to produce an executable workflow. The workflow is invoked to generate the migrating transformation, codified in the Atlas Transformation Language (ATL) [Jouault & Kurtev 2005]. Devising correct matching strategies was difficult, as AML lacks documentation that describes the input, output and effects of each heuristic. Papers describing AML (such as [Garcés *et al.* 2009]) discuss each heuristic, but mostly in a high-level manner. A semantically invalid combination of heuristics can cause a runtime error, while an incorrect combination results in the generation of an incorrect migration transformation. However, once a matching strategy is specified, it can be re-used for similar cases of metamodel evolution. To devise the matching strategies used in this paper, AML's author provided considerable guidance.

COPE. A COPE user applies *coupled operations* to the original metamodel to form the evolved metamodel. Each coupled operation specifies a metamodel evolution along with a corresponding fragment of the model migration strategy.

[4] `http://github.com/louismrose/migration_comparison`

A history of applied operations is later used to generate a complete migration strategy. As COPE is meant for co-evolution of models and metamodels, reverse engineering a large metamodel can be difficult. Determining which sequence of operations will produce a correct migration is not always straightforward. To aid the user, COPE allows operations to be undone. To help with the migration process, COPE offers the *Convergence View* which utilises EMF Compare to display the differences between two metamodels. While this was useful, it can, understandably, only provide a list of explicit differences and not the semantics of a metamodel change. Consequently, reverse-engineering a large and unfamiliar metamodel is challenging, and migration for the GMF Graph example could only be completed with considerable guidance from the author of COPE.

Ecore2Ecore. In Ecore2Ecore model migration is specified in two steps. In the first step, a graphical mapping editor is used to construct a model that declares basic migrations. In this step only very simple migrations such as class and feature renaming can be declared. In the next step, the developer needs to use Java to specify a customised parser (resource handler, in EMF terminology) that can parse models that conform to the original metamodel and migrate them so that they conform to the new metamodel. This customised parser exploits the basic migration information specified in the first step and delegates any changes that it cannot recognise to a particular Java method in the parser for the developer to handle. Handling such changes is tedious as the developer is only provided with the string contents of the unrecognised features and then needs to use low-level techniques – such as data-type checking and conversion, string splitting and concatenation – to address them. Here it is worth mentioning that Ecore2Ecore cannot handle all migration scenarios and is limited to cases where only a certain degree of structural change has been introduced between the original and the evolved metamodel. For cases which Ecore2Ecore cannot handle, developers need to specify a custom parser without any support for automated element copying.

Flock. In Flock, model migration is specified manually. Flock automatically copies only those model elements which still conform to the evolved metamodel. Hence, the user specifies migration only for model elements which no longer conform to the evolved metamodel. Due to the automatic copying algorithm, an empty Flock migration strategy always yields a model conforming to the evolved metamodel. Consequently, a user typically starts with an empty migration strategy and iteratively refines it to migrate non-conforming elements. However, there is no support to ensure that all non-conforming elements are migrated. In the GMF Graph example, completeness could only be ensured by testing with numerous models. Using this method, a migration strategy can be easily encoded for the Petri net example. For the GMF Graph example whose metamodels are larger, it was more difficult, since there is no tool support for analysing the changes between original and evolved metamodel.

Changing the migration strategy

Migration strategies can change in at least two ways. Firstly, as a migration strategy is developed, testing might reveal errors which need to be corrected.

Secondly, further metamodel changes might require changes to an existing migration strategy.

AML. Because AML automatically generates migrating transformations, changing the transformation, for example after discovering an error in the matching strategy, is trivial. To migrate models over several versions of a metamodel at once, the migrating transformations generated by AML can be composed by the user. AML provides no tool support for composing transformations.

COPE. As mentioned previously, COPE provides an undo feature, meaning that any incorrect migrations can be easily fixed. COPE stores a history of *releases* – a set of operations that has been applied between versions of the metamodel. Because the migration code generated from the release history can migrate models conforming to any previous metamodel release, COPE provides a comprehensive means for chaining migration strategies.

Ecore2Ecore. Migrations specified using Ecore2Ecore can be modified via the graphical mapping editor and the Java code in the custom model parser. Therefore, developers can use the features of the Eclipse Java IDE to modify and debug migrations. Ecore2Ecore provides no tool support for composing migrations, but composition can be achieved by modifying the resource handler.

Flock. There is comprehensive support for fixing errors. A migration strategy can easily be re-executed using a launch configuration, and migration errors are linked to the line in the migration strategy that caused the error to occur. If the metamodel is further evolved, the original migration strategy has to be extended, since there is no explicit support to chain migration strategies. The full migration strategy may need to be read to know where to extend it.

Extensibility

The fundamental constructs used for specifying migration in COPE and AML (operators and match heuristics, respectively) are extensible. Flock and Ecore2Ecore use a more imperative (rather than declarative) approach, and as such do not provide extensible constructs.

AML. An AML user can specify additional matching heuristics. This requires understanding of AML's domain-specific language for manipulating the data structures from which migrating transformations are generated.

COPE provides the user with a large number of operations. If there is no applicable operation, a COPE user can write their own operations using an in-place transformation language embedded into Groovy[5].

[5] http://groovy.codehaus.org/

Re-use

Each migration tool capture patterns that commonly occur in model migration. This section considers the extent to which the patterns captured by each tool facilitate re-use between migration strategies.

AML. Once a matching strategy is specified, it can potentially be re-used for further cases of metamodel evolution. Match heuristics provide a re-usable and extensible mechanism for capturing metamodel change and model migration patterns.

COPE. An operation in COPE represents a commonly occurring pattern in metamodel migration. Each operation captures the metamodel evolution and model migration steps. Custom operations can be written and re-used.

Ecore2Ecore. Mapping models cannot be reused or extended in Ecore2Ecore but as the custom model parser is specified in Java, developers can decompose it into reusable parts some of which can potentially be reused in other migrations.

Flock. A migration strategy encoded in Flock is modularised according to the classes whose instances need migration. There is support to reuse code within a strategy by means of operations with parameters and across strategies by means of imports. Re-use in Flock captures only migration patterns, and not the higher level co-evolution patterns captured in COPE or AML.

Conciseness

A concise migration strategy is arguably more readable and requires less effort to write than a verbose migration strategy. This section comments on the conciseness of migration strategies produced with each tool, and reports the lines of code (without comments and blank lines) used.

AML. 117 lines were automatically generated for the Petri nets example. 563 lines were automatically generated for the GMF Graph example, and a further 63 lines of code were added by hand to complete the transformation. Approximately 10 lines of the user-defined code could be removed by restructuring the generated transformation.

COPE requires the user to apply operations. Each operation application generates one line of code. The user may also write additional migration code. For the Petri net example, 11 operations were required to create the migrator and no additional code. The author of COPE migrated the GMF Graph example using 76 operations and 73 lines of additional code.

Ecore2Ecore. As discussed above, handling changes that cannot be declared in the mapping model is a tedious task and involves a significant amount of low level code. For the PetriNets example, the Ecore2Ecore solution involved a mapping model containing 57 lines of (automatically generated) XMI and a custom hand-written resource handler containing 78 lines of Java code.

Flock. 16 lines of code were necessary to encode the Petri nets example, and 140 lines of code were necessary to encode the GMF Graph example. In the GMF Graph example, approximately 60 lines of code implement missing built-in support for rule inheritance, even after duplication was removed by extracting and re-using a subroutine.

Clarity

Because migration strategies can change and might serve as documentation for the history of a metamodel, their clarity is important. This section reports on aspects of each tool that might affect the clarity of migration strategies.

AML. The AML code generator takes a conservative approach to naming variables, to minimise the chances of duplicate variable names. Hence, some of the generated code can be difficult to read and hard to re-use if the generated transformation has to be completed by hand. When a complete transformation can be generated by AML, clarity is not as important.

COPE. Migration strategies in COPE are defined as a sequence of operations. The release history stores the set of operations that have been applied, so the user is clearly able to see the changes they have made, and find where any issues may have been introduced.

Ecore2Ecore. The graphical mapping editor provided by Ecore2Ecore allows developers to have a high-level visual overview of the simple mappings involved in the migration. However, migrations expressed in the Java part of the solution can be far more obscure and difficult to understand as they mix high-level intention with low-level string management operations.

Flock clearly states the migration strategy from the source to the target metamodel. However, the boilerplate code necessary to implement rule inheritance slightly obfuscates the real migration code.

Expressiveness

Migration strategies are easier to infer for some categories of metamodel change than others [Gruschko *et al.* 2007]. This section reports on the ability of each tool to migrate the examples considered in this comparison.

AML. A complete migrating transformation could be generated for the Petri nets example, but not for the GMF Graph example. The latter contains examples of two complex changes that AML does not currently support[6]. Successfully expressing the GMF Graph example in AML would require changes to at least one of AML's heuristics. However, AML provided an initial migration transformation that was completed by hand.

In general, AML cannot be used to generate complete migration strategies for co-evolution examples that contain *breaking and non-resolvable changes*, according to the categorisation proposed in [Gruschko *et al.* 2007].

[6] `http://www.eclipse.org/forums/index.php?t=rview&goto=526894#msg_526894If`

COPE. The expressiveness of COPE is defined by the set of operations available. The Petri net example was migrated using only built-in operations. The GMF Graph example was migrated using 76 built-in operations and 2 user-defined migration actions. Custom migration actions allow users to specify any migration strategy.

Ecore2Ecore. A complete migration strategy could be generated for the Petri nets example, but not for the GMF Graph example. The developers of Ecore2Ecore have advised that the latter involves significant structural changes between the two versions and recommended implementing a custom model parser from scratch.

Flock. Since Flock extends EOL, it is expressive enough to encode both examples. However, Flock does not provide an explicit construct to copy model elements and thus it was necessary to call Java code from within Flock for the GMF Graph example.

Interoperability

Migration occurs in a variety of settings with differing requirements. This section considers the technical dependencies and procedural assumptions of each tool, and seeks to answer questions such as: "Which modelling technologies can be used?" and "What assumptions does the tool make on the migration process?"

AML depends only on ATL, while its development tools also require Eclipse. AML assumes that the original and target metamodels are available for comparison, and does not require a record of metamodel changes. AML can be used with either Ecore (EMF) or KM3 metamodels.

COPE depends on EMF and Groovy, while its development tools also require Eclipse and EMF Compare. COPE does not require both the original and target metamodels to be available. When COPE is used to create a migration strategy after metamodel evolution has already occurred, the metamodel changes must be reverse-engineered. To facilitate this, the target metamodel can be used with the Convergence View, as discussed in Section 6.3.2. COPE targets EMF, and does not support other modelling technologies.

Ecore2Ecore depends only on EMF. Both the original and the evolved versions of the metamodel are required to specify the mapping model with the Ecore2Ecore development tools. Alternatively, the Ecore2Ecore mapping model can be constructed programmatically and without using the original metamodel[7]. Unlike the other tools considered, Ecore2Ecore does not require the original metamodel to be available in the workspace of the metamodel user.

[7] Private communication with Marcelo Paternostro, an Ecore2Ecore developers.

Flock depends on Epsilon and its development tools also require Eclipse. Flock assumes that the original and target metamodels are available for encoding the migration strategy, and does not require a record of metamodel changes. Flock can be be used to migrate models represented in EMF, MDR, XML and Z (CZT), although we only encoded a migration strategy for EMF metamodels in the presented examples.

Performance

The time taken to execute model migration is important, particularly once a migration strategy has been distributed to metamodel users. Ideally, migration tools will produce migration strategies whose execution time is quick and scales well with large models.

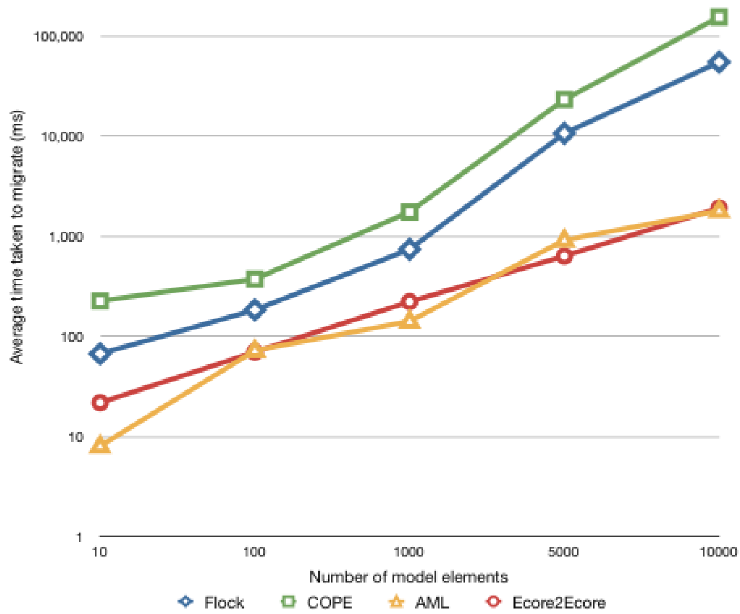


Figure 6.7: Migration tool performance comparison.

To measure performance, five sets of Petri net models were generated at random. Models in each set contained 10, 100, 1000, 5,000, and 10,000 model elements. Figure 6.7 shows the average time taken by each tool to execute migration across 10 repetitions for models of different sizes. Note that the Y axis has a logarithmic scale. The results indicate that, for the Petri nets co-evolution example, AML and Ecore2Ecore execute migration significantly more quickly than COPE and Flock, particularly when the model to be migrated contains more than 1,000 model elements. Figure 6.7 indicates that, for the Petri nets co-evolution example, Flock executes migration between two and three times faster than COPE, although the author of COPE reports that turning off validation causes COPE to perform similarly to Flock.

6.3.3 Discussion

The comparison described above highlights similarities and differences between a representative sample of model migration approaches. From this comparison, guidance for selecting between tools was synthesised. The guidance is presented below, and was produced by all four participants in the comparison (Herrmannsdöerfer, Williams, Kolovos and myself).

COPE captures co-evolution patterns (which apply to both model and metamodel), while Ecore2Ecore, AML and Flock capture only model migration patterns (which apply just to models). Because of this, COPE facilitates a greater degree of re-use in model migration than other approaches. However, the order in which the user applies patterns with COPE impacts on both metamodel evolution and model migration, which can complicate pattern selection particularly when a large amount of evolution occurs at once. The re-usable co-evolution patterns in COPE make it well suited to migration problems in which metamodel evolution is frequent and in small steps.

Flock, AML and Ecore2Ecore are preferable to COPE when metamodel evolution has occurred before the selection of a migration approach. Because of its use of co-evolution patterns, we conclude that COPE is better suited to forward- rather than reverse-engineering.

Through its Convergence View and integration with the EMF metamodel editor, COPE facilitates metamodel analysis that is not possible with the other approaches considered in this paper. COPE is well-suited to situations in which measuring and reasoning about co-evolution is important.

In situations where migration involves modelling technologies other than EMF, AML and Flock are preferable to COPE and Ecore2Ecore. AML can be used with models represented in KM3, while Flock can be used with models represented in MDR, XML and CZT. Via the connectivity layer of Epsilon, Flock can be extended to support further modelling technologies.

There are situations in which Ecore2Ecore or AML might be preferable to Flock and COPE. For large models, Ecore2Ecore and AML might execute migration significantly more quickly than Flock and COPE. Ecore2Ecore is the only tool that has no technical dependencies (other than a modelling framework). In situations where migration must be embedded in another tool, Ecore2Ecore offers a smaller footprint than other migration approaches. Compared to the other approaches considered in this paper, AML automatically generates migration strategies with the least guidance from the user.

Despite these advantages, Ecore2Ecore and AML are unsuitable for some types of migration problem, because they are less expressive than Flock and COPE. Specifically, changes to the containment of model elements typically cannot be expressed with Ecore2Ecore and changes that are classified by [Herrmannsdoerfer *et al.* 2008] as *metamodel-specific* cannot be expressed with AML. Because of this, it is important to investigate metamodel changes before selecting a migration tool. Furthermore, it might be necessary to anticipate which types of metamodel change are likely to arise before selecting a migration tool. Investing in one tool to discover later that it is no longer suitable causes wasted effort.

Table 6.4: Summary of tool selection advice. (Tools are ordered alphabetically).

Requirement	**Recommended Tools**
Frequent, incremental co-evolution	COPE
Reverse-engineering	AML, Ecore2Ecore, Flock
Modelling technology diversity	Flock
Quicker migration for larger models	AML, Ecore2Ecore
Minimal dependencies	Ecore2Ecore
Minimal hand-written code	AML, COPE
Minimal guidance from user	AML
Support for metamodel-specific migrations	COPE, Flock

6.3.4 Summary

The work presented in this section compared a representative sample of approaches to automating model migration. The comparison was performed by following a methodical process and using an example from a real-world MDE project. Some preliminary recommendations and guidelines in choosing a migration tool were synthesised from the presented results and are summarised in Table 6.4. The comparison was carried out by the developers of the migration tools (or stand-ins where the developers were unable to participate fully). Each developer used a tool other than their own so that the comparison could more closely emulate the level of expertise of a typical user.

Some criteria were excluded from the comparison because of the method employed. For instance, the learnability of a tool affects the productivity of users, and, as such, affects tool selection. However, drawing conclusions about learnability (and also productivity and usability) is challenging with the comparison method employed because of the subjective nature of these characteristics. A comprehensive user study (with hundreds of users) would be more suitable for assessing these types of criteria.

6.4 Transformation Tools Contest

The Transformation Tools Contest (TTC) is a workshop series that seeks to compare and contrast tools for performing model and graph transformation. At TTC 2010, two rounds of submissions were invited: cases (transformation problems, three of which are selected by the workshop organisers) and solutions to the selected cases. In addition, TTC 2010 include a *live contest*: during the workshop a further transformation problem was announced and solutions submitted.

Participation in TTC 2010 facilitated further evaluation of Flock. Flock and 8 other transformation tools were assessed for a model migration problem based on a real-world example of metamodel evolution from the UML [OMG 2007b]. As part of the live contest, Flock was also assessed along with 13 transformation tools for a model transformation problem. Compared to the evaluation described in Section 6.3, the evaluation in this section compares Flock to a wider range of tools (model and graph transformation tools, and not just model migration tools), and investigates the suitability of Flock for specifying model transformation (and not just model migration).

The remainder of this section describes the model migration problem (Section 6.4.1) and Flock solution (Section 6.4.2), and the use of Flock for specifying a model transformation in the live contest (Section 6.4.3).

6.4.1 Model Migration Case

To compare Flock with other transformation tools for specifying model migration, the thesis author submitted a case to TTC based on the evolution of the UML. The way in which activity diagrams are modelled in the UML changed significantly between versions 1.4 and 2.1 of the specification. In the former, activities were defined as a special case of state machines, while in the latter they are defined atop a more general semantic base[8] [Selic 2005].

The remainder of this section briefly introduces UML activity diagrams, describes their evolution, and discusses the way in which solutions were assessed. The work presented in this section is based on the case submitted to TTC 2010 [Rose *et al.* 2010d].

Activity Diagrams in UML

Activity diagrams are used for modelling lower-level behaviours, emphasising sequencing and co-ordination conditions. They are used to model business processes and logic [OMG 2007b]. Figure 6.8 shows an activity diagram for filling orders. The diagrams is partitioned into three *swimlanes*, representing different organisational units. *Activities* are represented with rounded rectangles and *transitions* with directed arrows. *Fork* and *join* nodes are specified using a solid black rectangle. *Decision* nodes are represented with a diamond. Guards on transitions are specified using square brackets. For example, in Figure 6.8 the transition to the restock activity is guarded by the condition `[not in stock]`. Text on transitions that is not enclosed in square brackets represents a trigger event. In Figure 6.8, the transition from the restock activity occurs on receipt of the asynchronous signal called `receive stock`. Finally, the transitions between activities might involve interaction with objects. In Figure 6.8, the Fill Order activity leads to an interaction with an object called `Filled Object`.

Between versions 1.4 and 2.2 of the UML specification, the metamodel for activity diagrams has changed significantly. The sequel summarises most of the changes, and details can be found in [OMG 2001] and [OMG 2007b].

Evolution of Activity Diagrams

Figures 6.9 and 6.10 are simplifications of the activity diagram metamodels from versions 1.4 and 2.2 of the UML specification, respectively. In the interest of clarity, some features and abstract classes have been removed from Figures 6.9 and 6.10.

Some differences between Figures 6.9 and 6.10 are: activities have been changed such that they comprise nodes and edges, actions replace states in UML 2.2, and the subtypes of control node replace pseudostates.

[8] A variant of generalised coloured Petri nets.

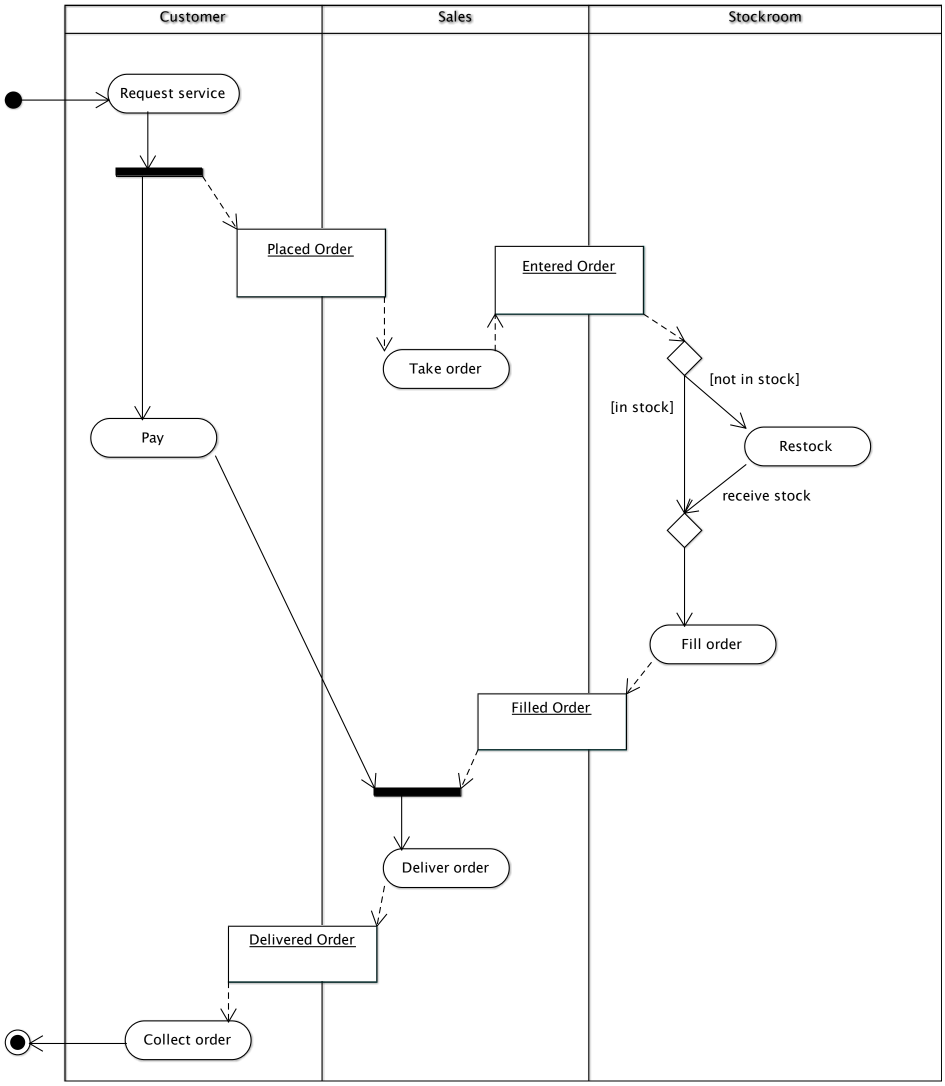


Figure 6.8: Activity model to be migrated.

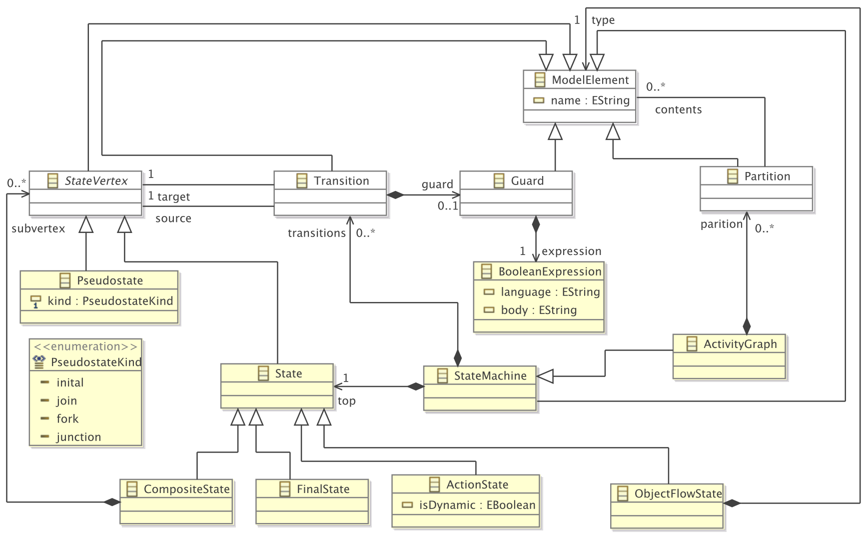


Figure 6.9: UML 1.4 Activity Graphs (based on [OMG 2001]).

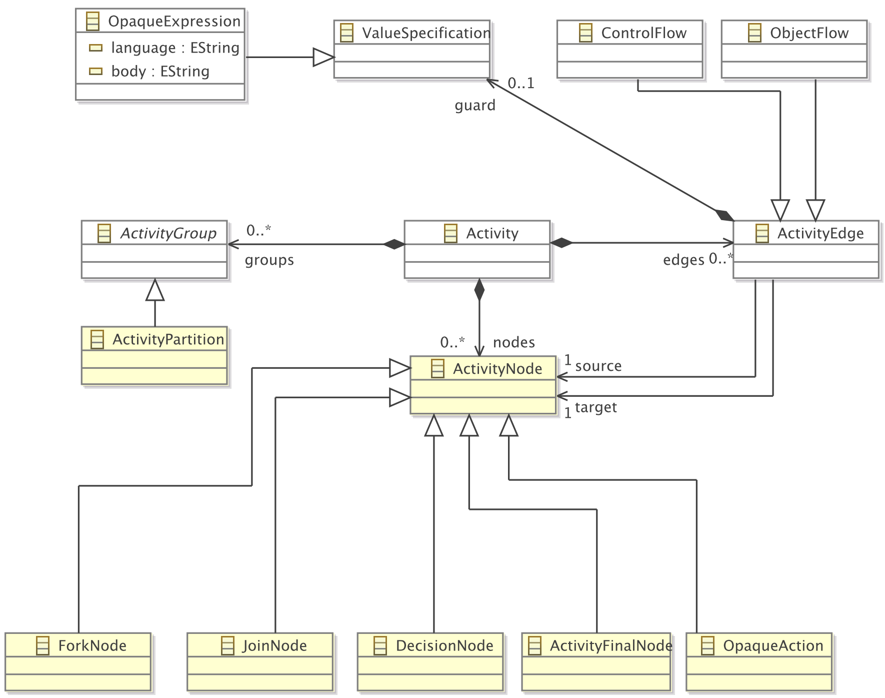


Figure 6.10: UML 2.2 Activity Diagrams (based on [OMG 2007b]).

To facilitate the comparison of solutions, the exemplar model shown in Figure 6.8 was used. Figure 6.8 is based on [OMG 2001, pg3-165]. Solutions migrated the activity diagram shown in Figure 6.8 – which conforms to UML 1.4 – to conform to UML 2.2. The UML 1.4 model, the migrated UML 2.2 model, and the UML 1.4 and 2.2 metamodels are available from[9].

Submissions were evaluated using the following four criteria, which were decided by the thesis author and the workshop organisers:

- **Correctness**: Does the transformation produce a model equivalent to the migrated UML 2.2. model included in the case resources?
- **Conciseness**: How much code is required to specify the transformation? (In [Sprinkle & Karsai 2004] et al. propose that the amount of effort required to codify migration should be directly proportional to the number of changes between original and evolved metamodel).
- **Clarity**: How easy is it to read and understand the transformation? (For example, is a well-known or standardised language?)
- **Extensions**: Which of the case extensions (described below) were implemented in the solution?

To further distinguish between solutions, three extensions to the core task were proposed. The first extension was added after the case was submitted, and was proposed by the workshop organisers and the solution authors. The second and third extension were included in the case by the thesis author.

Extension 1: Alternative Object Flow State Migration Semantics

Following the submission of the case, discussion on the TTC forums[10] revealed an ambiguity in the UML 2.2 specification indicating that the migration semantics for the ObjectFlowState UML 1.4 concept are not clear from the UML 2.2 specification.

In the core task described above, instances of ObjectFlowState were migrated to instances of ObjectNode. Any instances of Transition that had an ObjectFlowState as their source or target were migrated to instances of ObjectFlow. Listing 6.7 shows an example application of this migration semantics. The top line of Listing 6.7 shows instances of UML 1.4 metaclasses, include an instance of ObjectFlowState. The bottom line of Listing 6.7 shows the equivalent UML 2.2 instances according to this migration semantics. Note that the Transitions, t1 and t2, are migrated to an instance of ObjectFlow. Likewise, the instance of ObjectFlowState, s2, is migrated to an instance of ObjectFlow.

```
s1:State <- t1:Transition -> s2:ObjectFlowState <- t2:Transition -> s3:State

s1:ActivityNode <- t1:ObjectFlow -> s2:ObjectNode <- t2:ObjectFlow -> s3:
    ActivityNode
```

Listing 6.7: Migrating Actions

[9] http://www.cs.york.ac.uk/~louis/ttc/

[10] http://planet-research20.org/ttc2010/index.php?option=com_community&view=groups&task=viewgroup&groupid=4&Itemid=150 (registration required)

This extension considered an alternative migration semantics for ObjectFlowState. For this extension, instances of ObjectFlowState (and any connected Transitions) were migrated to instances ObjectFlow, as shown by the example in Listing 6.8 in which the UML 2.2 ObjectFlow, f1, replaces t1, t2 and s2.

```
s1:State <- t1:Transition -> s2:ObjectFlowState <- t2:Transition -> s3:State

s1:ActivityNode <- f1:ObjectFlow -> s3:ActivityNode
```

Listing 6.8: Migrating Actions

The alternative semantics were proposed on the TTC 2010 forums, and agreed as an extension to the core task by consensus between the solution authors and the workshop organisers.

Extension 2: Concrete Syntax

The second extension relates to the appearance of activity diagrams. The UML specifications provide no formally defined metamodel for the concrete syntax of UML diagrams. However, some UML tools store diagrammatic information in a structured manner using XML or a modelling tool. For example, the Eclipse UML 2 tools [Eclipse 2009b] store diagrams as GMF [Gronback 2009] diagram models.

As such, submissions were invited to explore the feasibility of migrating the concrete syntax of the activity diagram shown in Figure 6.8 to the concrete syntax in their chosen UML 2 tool. To facilitate this, the case resources included an ArgoUML project[11] containing the activity diagram shown in Figure 6.8.

Extension 3: XMI

The UML specifications indicate that UML models should be stored using XMI. However, because XMI has evolved at the same time as UML, UML 1.4 tools most likely produce XMI of a different version to UML 2.2 tools. For instance, ArgoUML produces XMI 1.2 for UML 1.4 models, while the Eclipse UML2 tools produce XMI 2.1 for UML 2.2.

As an extension to the core task, submissions were invited to consider how to migrate a UML 1.4 model represented in XMI 1.x to a UML 2.1 model represented in XMI 2.x. To facilitate this, the UML 1.4 model shown in Figure 6.8 was made available in XMI 1.2 as part of the case resources.

Following the submission of the case, solutions were encouraged to solve this extension by Tom Morris, the project leader for ArgoEclipse and a committer on ArgoUML. On the TTC forums, Morris stated that "We have nothing available to fill this hole currently, so any contributions would be hugely valuable. Not only would achieve academic fame and glory from the contest, but you'd get to see your code benefit users of one of the oldest (10+ yrs) open source UML modeling tools." [12]

[11] http://argouml.tigris.org/

[12] http://www.planet-research20.org/ttc2010/index.php?option=com_community&view=groups&task=viewdiscussion&groupid=4&topicid=20&Itemid=150 (registration required)

6.4.2 Model Migration Solution in Epsilon Flock

This section discusses the Flock solution to the TTC case described above (the evolution of UML activity diagrams).

The solution was developed in an iterative and incremental manner, using the following process:

1. Change the Flock migration strategy.
2. Execute Flock on the original model, producing a migrated model.
3. Compare the migrated model with the reference model provided in the case resources.
4. Repeat until the migrated and reference models were the same.

The remainder of this section presents the Flock solution in an incremental manner. The code listings in this section show only those rules relevant to the iteration being discussed.

Actions, Transitions and Final States

Development of the migration strategy began by executing an empty Flock migration strategy on the original model. Because Flock automatically copies model elements that have not been affected by evolution, the resulting model contained `Pseudostatess` and `Transitions`, but none of the `ActionStates` from the original model. In UML 2.2 activities, `OpaqueActions` replace `ActionStates`. Listing 6.9 shows the Flock code for changing `ActionStates` to corresponding `OpaqueActions`.

```
migrate ActionState to OpaqueAction
```

Listing 6.9: Migrating Actions

Next, similar rules were added to migrate instances of `FinalState` to instances of `ActivityFinalNode` and to migrate instances of `Transition` to `ControlFlow`, as shown in Listing 6.10.

```
migrate FinalState to ActivityFinalNode
migrate Transition to ControlFlow
```

Listing 6.10: Migrating FinalStates and Transitions

Pseudostates

Development continued by selected further types of state that were not present in the migrated model, such as `Pseudostatess`, which are not used in UML 2.2 activities. Instead, UML 2.2 activities use specialised `Nodes`, such as `InitialNode`. Listing 6.11 shows the Flock code used to change `Pseudostates` to corresponding `Nodes`.

```
migrate Pseudostate to InitialNode when: original.kind = Original!
      PseudostateKind#initial
migrate Pseudostate to DecisionNode when: original.kind = Original!
      PseudostateKind#junction
migrate Pseudostate to ForkNode when: original.kind = Original!
      PseudostateKind#fork
migrate Pseudostate to JoinNode when: original.kind = Original!
      PseudostateKind#join
```

Listing 6.11: Migrating Pseudostates

Activities

In UML 2.2, `Activity`s no longer inherit from state machines. As such, some of the features defined by `Activity` have been renamed. Specifically, `transitions` has become `edges` and `paritions` has become `group`. Furthermore, the states (or nodes in UML 2.2 parlance) of an `Activity` are now contained in a feature called `nodes`, rather than in the `subvertex` feature of a composite state accessed via the `top` feature of `Activity`. The Flock migration rule shown in Listing 6.12 captured these changes.

```
migrate ActivityGraph to Activity {
  migrated.edge = original.transitions.equivalent();
  migrated.group = original.partition.equivalent();
  migrated.node = original.top.subvertex.equivalent();
}
```

Listing 6.12: Migrating ActivityGraphs

Note that the rule in Listing 6.12 used the built-in `equivalent` operation to find migrated model elements from original model elements. As discussed in Section 5.4, the `equivalent` operation invokes other migration rules where necessary and caches results to improve performance.

Next, a similar rule for migrating `Guard`s was added. In UML 1.4, the the `guard` feature of `Transition` references a `Guard`, which in turn references an `Expression` via its `expression` feature. In UML 2.2, the `guard` feature of `Transition` references an `OpaqueExpression` directly. Listing 6.13 captures this in Flock.

```
migrate Guard to OpaqueExpression {
  migrated.body.add(original.expression.body);
}
```

Listing 6.13: Migrating Guards

Partitions

In UML 1.4 activity diagrams, `Partition` specifies a single containment reference for its `contents`. In UML 2.2 activity diagrams, partitions have been renamed to `ActivityPartitions` and specify two containment features for their contents, `edges` and `nodes`. Listing 6.14 shows the rule used to migrate `Partitions` to `ActivityPartitions` in Flock. The body of the rule shown in Listing 6.14 uses the *collect* operation to segregate the `contents` feature of the original model element into two parts.

```
migrate Partition to ActivityPartition {
  migrated.edges = original.contents.collect(e:Transition | e.equivalent());
  migrated.nodes = original.contents.collect(n:StateVertex | n.equivalent())
      ;
}
```

Listing 6.14: Migrating Partitions

ObjectFlows

Finally, two rules were written for migrating model elements relating to object flows. In UML 1.4 activity diagrams, object flows are specified using `ObjectFlowState`, a subtype of `StateVertex`. In UML 2.2 activity diagrams, object flows are modelled using a subtype of `ObjectNode`. In UML

2.2 flows that connect to and from ObjectNodes must be represented with ObjectFlows rather than ControlFlows.

Listing 6.15 shows the Flock rule used to migrate Transitons to ObjectFlows. The rule applies for Transitions whose source or target StateVertex is of type ObjectFlowState.

```
migrate ObjectFlowState to ActivityParameterNode

migrate Transition to ObjectFlow when: original.source.isTypeOf(
    ObjectFlowState) or original.target.isTypeOf(ObjectFlowState)
```

Listing 6.15: Migrating ObjectFlows

In addition to the core task, the Flock solution also approached two of the three extensions described in the case (Section 6.4.1). The solutions to the extensions are now discussed.

Alternative ObjectFlowState Migration Semantics

The first extension required submissions to consider an alternative migration semantics for ObjectFlowState, in which a single ObjectFlow replaces each ObjectFlowState and any connected Transitions.

Listing 6.16 shows the Flock source code used to migrate ObjectFlowStates (and connecting Transitions) to a single ObjectFlow. This rule was used instead of the two rules defined in Listing 6.15. In the body of the rule shown in Listing 6.16, the source of the Transition is copied directly to the source of the ObjectFlow. The target of the ObjectFlow is set to the target of the first outgoing Transition from the ObjectFlowState.

```
migrate Transition to ObjectFlow when: original.target.isTypeOf(
    ObjectFlowState) {
  migrated.source = original.source.equivalent();
  migrated.target = original.target.outgoing.first.target.equivalent();
}
```

Listing 6.16: Migrating ObjectFlowStates to a single ObjectFlow

Because, in this alternative semantics, ObjectFlowStates are represented as edges rather than nodes, the partition migration rule was changed such that ObjectFlowStates were not copied to the nodes feature of Partitions. To filter out the ObjectFlowStates, line 3 of Listing 6.14 was changed to include a reject statement, as shown on line 3 of Listing 6.17.

```
1 migrate Partition to ActivityPartition {
2   migrated.edges = original.contents.collect(e:Transition | e.equivalent());
3   migrated.nodes = original.contents.reject(ofs:ObjectFlowState | true).
      collect(n:Original!StateVertex | n.equivalent());
4 }
```

Listing 6.17: Migrating Partitions without ObjectFlowStates

XMI

The second extension required submissions to migrate an activity graph conforming to UML 1.4 and encoded in XMI 1.2 to an equivalent activity graph conforming to UML 2.2 and encoded in XMI 2.1. The core task did not require submissions to consider changes to XMI (the model storage representation), but, in practice, this is a challenge to migration, as noted by Tom Morris on the TTC forums[13].

As discussed in Section 5.4, Flock is built atop Epsilon, which includes a model connectivity layer (EMC). EMC provides a common interface for accessing and persisting models. Currently, EMC supports EMF (XMI 2.x), MDR (XMI 1.x), and plain XML models. To support migration between metamodels defined in heterogenous modelling frameworks, EMC was extended during the development of Flock to provide a conformance checking service.

Consequently, the migration strategy developed for the core task works for all of the types of model supported by EMC. To migrate a model encoded in XMI 1.2 rather than in XMI 2.1, the user must select a different option when executing the Flock migration strategy. Otherwise, no other changes are required.

Results

At the workshop, solutions to the migration case described in Section 6.4.1 were presented. Each solution was allocated two opponents who highlighted weaknesses of each approach. Following the solution presentations and opposition statements, each solution was scored using the four criteria described above, correctness, clarity, conciseness and number of extensions solved. Every workshop participants scored each solution on clarity and conciseness. The workshop organisers scored each solution on correctness and number of extensions solved, as these criteria could be measured objectively. Epsilon Flock was awarded first position for the migration case.

The opposition statements highlighted two weaknesses of Flock. Firstly, there is some duplicated code in Listing 6.11: the `migrate Pseudostate to X` statement appears several times. The duplication exists because Flock only allows one-to-one mappings between original and evolved metamodel types. The conservative copy algorithm would need to be extended to allow one-to-many mappings to remove this kind of duplication.

Secondly, the body of Flock rules are specified in an imperative manner. Consequently, reasoning about the correctness of the a migration strategy is arguably more difficult than in languages that use a purely declarative syntax. This point is discussed further in Section 6.5, which considers the limitations of the thesis.

6.4.3 Epsilon Flock in the Live Contest

TTC 2010 also invited the workshop participants to take part in a live contest. A problem was announced at the start of the workshop, and participants developed their solutions during the first day. The solutions were presented in the workshop and assessed in four categories. Flock was awarded first position for the *exogenous transformation* category. The remainder of this section discusses the parts of the problem that relate to the exogenous transformation category and the Flock solution.

[13] `http://www.planet-research20.org/ttc2010/index.php?option=com_community&view=groups&task=viewdiscussion&groupid=4&topicid=20&Itemid=150` (registration required)

The live contest problem required several model management operations be combined to perform beta-reduction of a simplified lambda calculus. Flock was used to specify one of the model management operations: model transformation between two similar (but not identical) metamodels. Flock was chosen rather than a new-target transformation language because the metamodels shared several classes and features. Using Flock allowed automatic copying of the model elements conforming to the classes common to both source and target metamodel. In other words, transformation rules were specified only for those parts of the metamodels that differed.

Flock was awarded first position by the workshop participants and organisers for the category in which it was entered. Participation in the live contest highlighted that, in addition to model migration, Flock can be used for specifying model transformation. In particular, Flock was appropriate because the source and target metamodels were similar (having several classes and features in common) and the conservative copy strategy reduced the number of rules required to specify the transformation.

6.4.4 Summary

This section has discussed the way in which Flock was evaluated by participating in the 2010 edition of the Transformation Tools Contest (TTC). Flock was assessed by application to an example of migration from the UML and comparison with eight other model and graph transformation tools. Flock was awarded first prize by the workshop participants and organisers. Additionally, Flock was used as part of a solution to a live contest developed during the workshop. The live contest highlighted that Flock is suitable for specifying some types of model transformation (in particular, those in which the source and target metamodel have common classes and features), as Flock was awarded first prize in the exogenous transformation category.

In addition to evaluating Flock, the work described in this section provides three further contributions. Firstly, the migration case submitted to TTC 2010, described in Section 6.4.1 provides a real-world example of co-evolution for use in future comparisons of model migration tools. The case is based on the evolution of UML, between versions 1.4 and 2.2. The migration strategy was devised by analysis of the UML specification, and by discussion between workshop participants.

Secondly, the Flock solution to the migration case (Section 6.4.2) demonstrates the way in which a migration strategy can be constructed using Flock. In particular, Section 6.4.2 describes an iterative and incremental development process and indicates that an empty Flock migration strategy can provide a useful starting point for development.

Finally, Section 5.4 claims that Flock support several modelling technologies. The solution described in Section 6.4.2 demonstrates the way in which Flock can be used to migrate models over two modelling technologies: MDR (XMI 1.x) and EMF (XMI 2.x), and hence supports the claim made in Section 5.4.

6.5 Limitations

The limitations and threats to the validity of the thesis research are now discussed. Some of the shortcomings identified here are elaborated on in Section 7.2, which highlights areas of future work.

Generality The thesis research focuses on model-metamodel co-evolution, but, as discussed in Chapter 4, metamodel changes can affect artefacts other than models. Model management operations and model editors are specified using metamodel concepts and, consequently, are affected when a metamodel changes. The work presented in Chapter 5 focuses on migrating models in response to metamodel changes, and does not consider integration with tools for migrating model management operations and model editors. To reduce the effort required to manage the effects of metamodel changes, it seems reasonable to envisage a unified approach that migrates models, model management operations, model editors, and other affected artefacts.

Reproducibility The analysis and evaluation presented in Chapters 4 and 6 respectively involved using migration tools to understand and assess their functionality. With the exceptions noted below, the work presented in these chapters is difficult to reproduce and therefore the results drawn are somewhat subjective. On the other hand, multiple approaches to analysis and evaluation have been taken, and the work has been published and subjected to peer review.

Not all of the work in Chapter 4 and 6 is difficult to reproduce. In particular, Section 4.2 describes limitations of existing migration tools and was derived from the experiments discussed in Appendix A. To aid reproducibility, evaluation methods are described in detail in Sections 6.2 and 6.3. In general, the lack of real-world examples of co-evolution restricts the extent to which any work in this area can be considered reproducible.

Formal semantics No formal semantics for the conservative copy algorithm (Section 5.4) have been provided. Instead, a reference implementation, Epsilon Flock, was developed, which facilitated comparison with other migration and transformation tools. Without a reference implementation, the evaluation described in Sections 6.2, 6.3 and 6.4 would have been impossible. For Epsilon as a whole, [Kolovos 2009] makes a similar case for choosing a reference implementation over a formal semantics. For domains where completeness and correctness are a primary concern, a formal semantics would be required before Flock could be applied to manage model-metamodel co-evolution.

6.6 Summary

To be completed, but will include a paragraph similar to the following:

In addition to the evaluation described in this chapter, the work presented in this thesis has been subjected to peer review by the academic and Eclipse communities. The thesis research has been published in papers at XX workshops, YY European conferences and ZZ international conferences. HUTN, Flock and Concordance (Chapter 5) are part of the Epsilon project, a member of the research incubator for the Eclipse Modeling Project (EMP), which is arguably the most active MDE community at present. EMP's research incubator hosts a limited number of participants, selected through a rigorous process and contributions made to the incubator undergo regular technical review.

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[Lehman 1978] Meir M. Lehman. Programs, cities, students - limits to growth? *Programming Methodology*, pages 42–62, 1978.

[Lehman 1980] Meir M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213–221, 1980.

[Lehman 1985] Meir M. Lehman. *Program evolution: processes of software change*. Academic, 1985.

[Lehman 1996] Meir M. Lehman. Laws of software evolution revisited. In *Proc. European Workshop on Software Process Technology*, pages 108–124, 1996.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[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.

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

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

[Pool 1997] R. Pool. *Beyond Engineering: How Society Shapes Technology*. Oxford University Press, 1997.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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