Evolution in Model-Driven Engineering

Louis M. Rose

April 16, 2010

Contents

Chapter 6

Evaluation

6.1 Evaluation Measures

6.1.1 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 tools 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 textual modelling notation described in Chapter 5.

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.

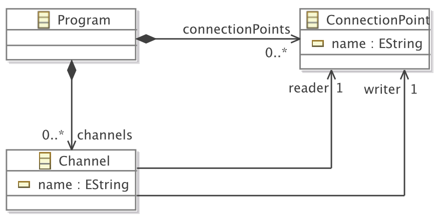
5. Check that the current (and all previous) graphical models are satisfactory representations of their hand-drawn counterparts.

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.

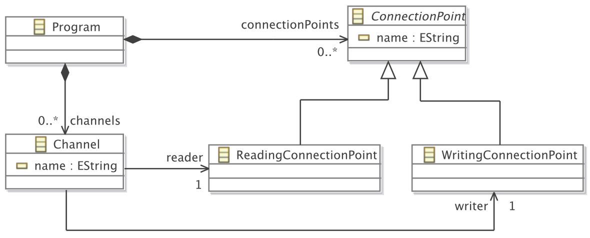
Here, one iteration of the process-oriented metamodel is considered, which led to the metamodel changes shown in Figure 6.1. In Figure 6.1(a), a `Program` is composed of `Channels` and `ConnectionPoints`. A `Channel` reads from and writes to exactly one `ConnectionPoint`. Further analysis of the domain 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).

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.



(a) Original metamodel.



(b) Evolved metamodel.

Figure 6.1: Process-oriented metamodel evolution.

Existing Process

The process-oriented metamodel was developed before the conformance reporting tool and textual modelling notation described in Chapter 5 were implemented. As such, model migration involved attempting to load existing, potentially non-conformant models with EMF, noting any conformance errors and changing the corresponding XMI to make the model conform to 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`, a rather technical process, which is discussed further below.

In a small number of cases, the wrong subtype of `ConnectionPoint` was selected, probably because XMI identifies elements using randomly generated strings rather than a domain-specific naming scheme. 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.

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 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[1]. 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."

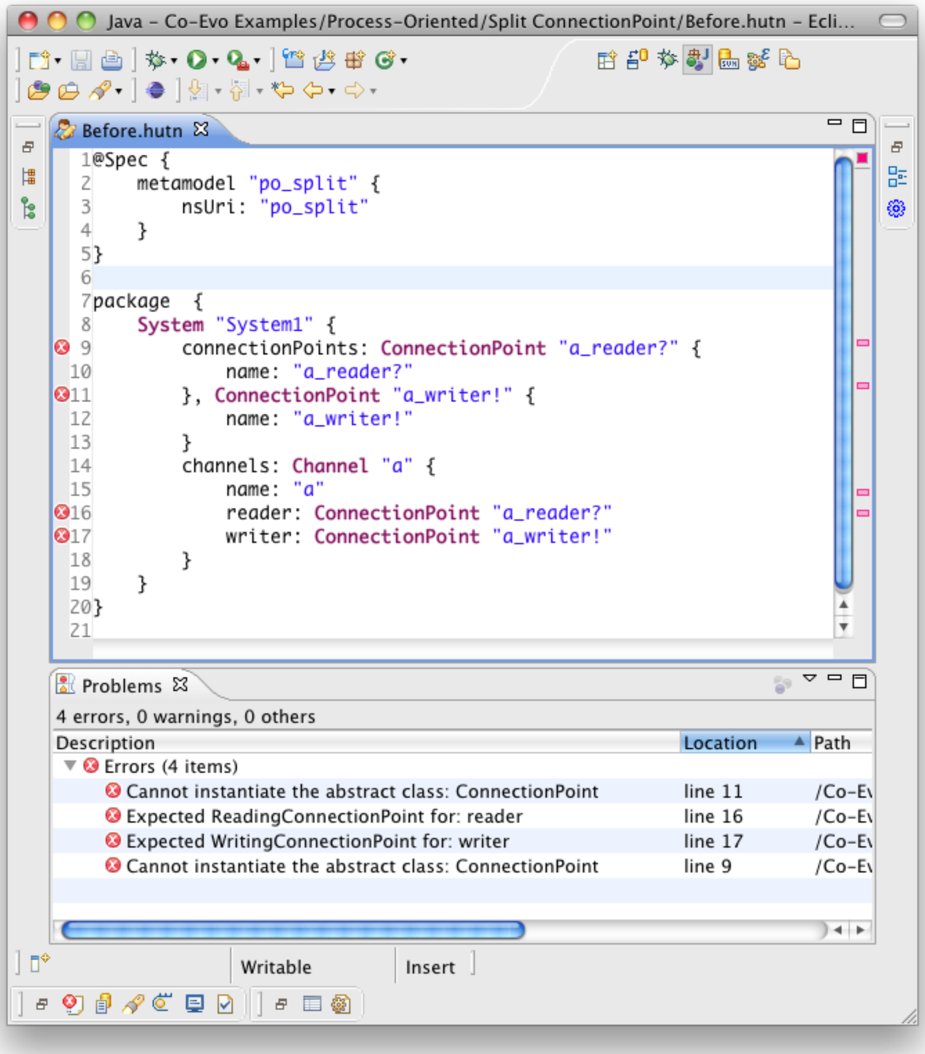
To fix the errors, a HUTN representation of each non-conformant model is generated by invoking the "Generate HUTN" context menu item. Figure 6.2 shows the HUTN generated for one of the non-conformant process-oriented models. Fixing the conformance problems involves changing the HUTN source by hand (and 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.

Summary

This section has demonstrated existing and new user-driven co-evolution processes using an example of metamodel changes. Comparison of the two highlights several benefits of the new 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 `Chan-`

[1]As described in Section 5.1.4, the conformance reporting tool can be invoked manually, via a context menu, or automatically as a result of integration with Concordance.


Java – Co-Evo Examples/Process-Oriented/Split ConnectionPoint/Before.hutn – Ecli...
Before.hutn
Problems
Writable
Insert

```
@Spec {
    metamodel "po_split" {
        nsUri: "po_split"
    }
}

package  {
    System "System1" {
        connectionPoints: ConnectionPoint "a_reader?" {
            name: "a_reader?"
        }, ConnectionPoint "a_writer!" {
            name: "a_writer!"
        }
        channels: Channel "a" {
            name: "a"
            reader: ConnectionPoint "a_reader?"
            writer: ConnectionPoint "a_writer!"
        }
    }
}
```

4 errors, 0 warnings, 0 others

Description	Location	Path
Errors (4 items)		
Cannot instantiate the abstract class: ConnectionPoint	line 11	/Co-Ev
Expected ReadingConnectionPoint for: reader	line 16	/Co-Ev
Expected WritingConnectionPoint for: writer	line 17	/Co-Ev
Cannot instantiate the abstract class: ConnectionPoint	line 9	/Co-Ev

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

nel#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 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 6.2 the Channel on line 14 refers to ConnectionPoints by name (lines 16 and 17). Hence, referencing and dereferencing 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.

6.1.2 Collaborative Case Study

6.1.3 Transformation Tools Contest

6.1.4 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 5.3.4, this thesis contributes Epsilon Flock, a domain-specific language for model migration. This section fulfils the second part of the above research requirement, 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]); general-purpose languages in others (e.g. [Herrmannsdoerfer *et al.* 2009, 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 6.1.4 describes the data used in the comparison. The method for the comparison is discussed in Section 6.1.4. Section 6.1.4 identifies model operations for each of the migration languages used in the comparison, and Section 6.1.4 presents and analysis the results.

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 identified in

Chapter 4, which were used to define requirements for Flock and conservative copy. 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 is 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 2006], 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 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[2].

The author is aware of two approaches to migration that use existing-target transformations. In COPE [Herrmannsdoerfer *et al.* 2009], migration strategies are hand-written in Groovy when no co-evolutionary operator can be applied. As discussed in Section 5.3.2, COPE's Groovy migration strategies use an existing-target approach. COPE provides six operations 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 operators 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[3] and cannot be used for migration in the co-evolution examples considered in this section.

Method

For each example of co-evolution, a migration strategy was written using each migration language (namely ATL, Groovy-for-COPE and Flock). The correctness of the migration strategy was assured by comparing the migrated models provided by the co-evolution example with the result of executing the migration strategy on the original models provided by the co-evolution example.

For each migration language, a set of model operations were identified, as described in Section 6.1.4. A program was written to count the number of

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

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

model operations appearing in each migration strategy. The counting program was tested by writing migration strategies in each language for the co-evolution examples identified in Chapter 4.

There is one non-trivial threat to the validity of the comparison performed in this section. The author wrote the migration strategies for Flock (a migration language that the author developed) and for the other migration languages considered (which the author has not developed). Therefore, it is possible that the migration strategies written in the latter 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.

Model Operations

The variation in frequency of model operations was explored across three model migration languages, ATL, Groovy-for-COPE and Flock. Here, the model operations of each language are identified. 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.

The comparison described in this section counts two categories of model operation: copying operations, deletion operations. The former are used to assign values to elements of the migrated model, while the latter are used to remove values from elements of the migrated model.

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

- Assignment to a feature:

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

Deletion 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.

TODO discuss whether it was possible to use AML to generate ATL and hence reduce the impact of the threat to validity identified above.

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

- Assignment to a feature:

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

- Unsetting a feature:

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

- Removing a model element:

  ```
  delete <element>
  ```

Deletion operations (unset and remove above) 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.

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. For example, the "Make Reference Containment" operator evolves the metamodel such that a non-containment reference becomes a containment reference and migrates models such that the values of the evolved reference are replaced by copies.

As such, writing the Groovy migration strategy for the examples of co-evolution considered in this section involved, where possible, applying an appropriate COPE co-evolutionary operator and counting the number of model operations in the generated migration strategy. Not all examples could be completely specified using COPE co-evolutionary operator. In these cases, the Groovy migration strategy was written by the author.

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.* 2006] to access and update model values. In addition, Flock defines `migrate` rules, which can be used to change the type of a model element. For Flock, the following model operations were counted:

- Assignment to a feature:

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

- Removing a model element:

  ```
  delete <element>
  ```

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. The latter is not necessary because conservative copy will never copy to the migrated model any value that does not conform the evolved metamodel.

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. Five examples of model migration were measured to obtain the results shown in Table 6.1.4. The results from measuring the examples identified from the analysis chapter are shown in Table 6.1.4.

Because the examples used to produce the measurements shown in Table 6.1.4 were used to design Flock, the measurements in Table 6.1.4 are less

	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).

	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).

relevant to the evaluation presented here than the measurements shown in Table 6.1.4. Nevertheless, the measurements made in Table 6.1.4 are included in the interest of transparency, and because they were used to test the program which performed the measurements.

For all but one of the examples shown above, conservative copy requires less model operations than new-target and existing-target. For the majority of examples, no migration strategy specified with existing-target contained less model operations when encoded with new-target. These 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 6.1.4, where ATL requires many more model operations than Groovy-for-COPE and Flock.

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 6.1.4) and Refactor Inline Class (Table 6.1.4) examples contained several feature renamings, and consequently the existing-target figure was nearer to the new-target figure than the conservative copy figure. This is contrary to the trend in Tables 6.1.4 and 6.1.4.

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 6.1.4 and 6.1.4.

Subtyping The GMF Graph example shown in Table 6.1.4 is the one case where conservative copy requires more model operations than existing-target. Investigating this result revealed a limitation in conservative copy limitation in Flock, relating to the way in subtypes are migrated.

Figure 6.3 shows a simplified part of the GMF Graph metamodel prior to evolution. When the metamodel evolved, the type 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 type of the `figure` and `accessor` features was 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 representative of the actual GMF Graph metamodel evolution.

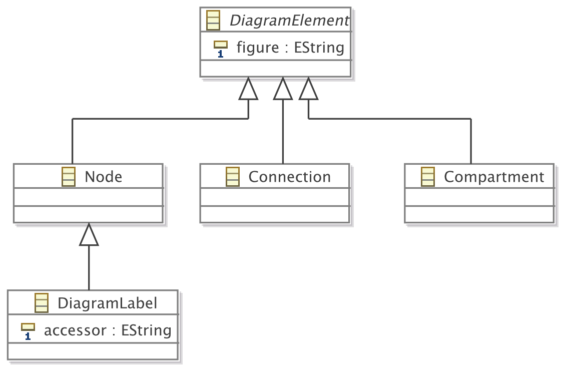


Figure 6.3: Simplified fragment of the GMF Graph metamodel.

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. For `DiagramLabels`, the values of both the accessor and figure feature must be migrated. On lines X-Y of Listing 6.1, the `DiagramLabels` extends `Nodes` and hence `DiagramElements` to inherit the body of the latter for migrating figures. 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
}

rule DiagramLabels extends Nodes {
 from o : Before!DiagramLabel
 to  m : After!DiagramLabel ( accessor <- o.accessor.length() )
}
```

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 in Listing 6.1. In Listing 6.2, the loop iterates over each instance of `DiagramElement`, 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()
3
4  if (subtyping.DiagramLabel.allInstances.contains(diagramElement))
       {
5   diagramElement.accessor = diagramElement.accessor.length()
6  }
7 }
```

Listing 6.2: Simplified GMF Graph model migration in COPE

In both ATL and 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 `migrate DiagramElement` rule cannot be used because the semantics of Flock migrate rules state that, when no to part is specified, Flock will create an instance of the type named after the keyword migrate (`DiagramElement` here). Because `DiagramElement` is abstract, this will fail. 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 in Flock, and in general, is discussed in Section 7.1. This section now considers one further limitation of existing-target and conservative copy, relative to new-target.

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`.

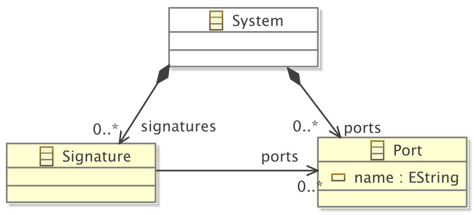


Figure 6.4: Original metamodel.

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.

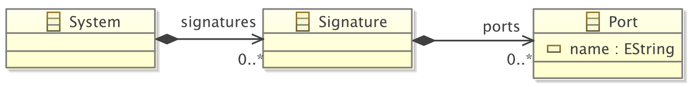


Figure 6.5: Evolved metamodel.

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, using a lazy rule, 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 existing-target migration languages, conformance is most likely only checked following the 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 corresponding 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, respectively, to automatically perform cloning when a reference is changed to be a containment reference. This is discussed further, for conservative copy, in Section 7.1.

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 thoroughly.

The analysis of the measurements obtained has shown that a new- and existing-target migration languages are most suitable for specifying migration strategies for different types of migration language. New-target requires less model operations than existing-target when metamodel evolution involves the renaming of features. Conversely, existing-target requires less model operations than new-target when metamodel evolution does not affect most model elements. Conservative copy requires less model operations than both new- and existing-target in almost all of the examples studied here.

This section has highlighted two limitations of the conservative copy algorithm implemented in Epsilon Flock, and 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, in the hope that similar comparisons might be conducted in the future.

6.2 Discussion

6.2.1 Threats to validity

6.3 Dissemination / Reception / ??

6.3.1 Publications

6.3.2 Delivery through Eclipse

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[Kolovos *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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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