Evolution in Model-Driven Engineering

Louis M. Rose

March 24, 2010

Contents

1 Introduction **5**
- 1.1 Model-Driven Engineering . . . 5
- 1.2 Software Evolution . . . 5
- 1.3 Research Aim . . . 5
- 1.4 Research Method . . . 5

2 Background **7**
- 2.1 Related Areas . . . 7
- 2.2 Categories of Evolution in MDE . . . 7

3 Literature Review **9**
- 3.1 Model-Driven Engineering . . . 9

4 Analysis **11**
- 4.1 Locating Data . . . 12
- 4.2 Analysing Existing Techniques . . . 18
- 4.3 Requirements Identification . . . 27
- 4.4 Chapter Summary . . . 30

5 Implementation **31**
- 5.1 Metamodel-Independent Syntax . . . 31
- 5.2 Textual Modelling Notation . . . 36
- 5.3 Epsilon Flock . . . 45
- 5.4 Chapter Summary . . . 55

6 Evaluation **57**
- 6.1 Evaluation Measures . . . 57
- 6.2 Discussion . . . 66
- 6.3 Dissemination / Reception / ?? . . . 66

7 Conclusion **53**

A Experiments **55**
- A.1 Metamodel-Independent Change . . . 55

Chapter 6

Evaluation

6.1 Evaluation Measures

6.1.1 Case Study

6.1.2 Collaborative Case Study

6.1.3 Transformation Tools Contest

6.1.4 Quantitive Comparison of Model Migration Languages

In Section **??**, 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. [**?**, **?**]); general-purpose languages in others (e.g. [**?**, **?**]). 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 **??**. 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. [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 TODO: GMFx2, Newsgroupsx2, UML Activities. Briefly describe these and explain where they came from.

These examples were not used in the work described in Chapters 4 and 5.

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 [**?**, **?**] for model migration. As discussed in Section **??**, model-to-model transformation languages support only new-target transformations for model migrationBecause, in model migration, the source and target metamodels are not the same..

The author is aware of two approaches to migration that use existing-target transformations. In COPE [**?**], migration strategies are hand-written in Groovy when no co-evolutionary operator can be applied. As discussed in Section **??**, COPE's Groovy migration strategies use an existing-target approach. COPE provides 6[2] 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 [**?**], migration is performed when the original model is loaded, effectively an existing-target approach. Ecore2Ecore migration strategies are written in Java and must interact with libraries for interacting with EMF and XML.

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

[1] TODO: Explain the structure of the rest of this section

[2] check this

ness 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 *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 Section **??**.

Model Migration Languages

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.

Atlas Transformation Language The Atlas Transformation Language (ATL), ...

TODO: Delete isn't measure because elements aren't deleted, they're simply not copied. TODO: Where do COPE / FLock use new? Why doesn't ATL? If it does, I need to count those too. OR, reconsider counting new - is it necessary?

- Assignment to a feature:

  `<element>.<feature> <- <value>`

- Changing the type of a model element:

`rule { from <name> : <original`$_{type}$ $>$ *to* $<$ *name* $>$:$<$ *migrated*$_{type}$ $>$
}

The `to` part of a `migrate` rule is optional. `Migrate` rules without `to` parts were not counted as model operations.

Groovy-for-COPE In COPE [**?**], migration strategies are codified in Groovy, a general-purpose, dynamically-typed programming language. COPE provides model operations for manipulating the migrated model via a metamodel-independent representation, as discussed in Chapters 4 and 5. 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()
```

- Creating a new model element:

```
<element_type>.newInstance()
```

- Deleting a model element:

```
delete <element>
```

- Changing the type of a model element:

```
<element>.migrate(<element_type>)
```

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>)
```

- Creating a new model element:

```
new <element_type>
```

- Deleting a model element:

```
delete <element>
```

- Changing the type of a model element:

`migrate` <original$_{t}$*ype* > *to* < *migrated*$_{t}$*ype* >

The `to` part of a `migrate` rule is optional. `Migrate` rules without `to` parts were not counted as model operations.

Table 6.1: Model operation frequency. An asterisk denotes an example that is not supported by Ecore2Ecore.

Name	ATL	COPE	Flock
Newsgroup Extract Person	9	7	6
Newsgroup Resolve Replies	8	3	2
UML Activity Diagrams	26	TBC	17
GMF Graph	101	TBC	16
GMF Gen2009	310	TBC	21
Totals	TBC	TBC	TBC
Averages	TBC	TBC	TBC

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.

TODO: might be interesting to reinstate the table for results from examples described in analysis chapter.

TODO: Comment on why GMF results are so much higher for ATL (It's because the source metamodels contain a lot of features. The UML example would have had similar differences in figures, but uses a minimal metamodel because ATL / COPE don't support MDR).

The results in Table 6.1 show that no migration strategy encoded in Flock contained less model operations when encoded in Groovy-for-COPE or ATL. For the majority of examples, no migration strategy encoded in Groovy-for-COPE contained less model operations when encoded in ETL. The reasons for the results shown in Table 6.1 are now investigated.

Copying Strategy Each approach initialises the migrated model in a different way: ATL initialises an empty model, while COPE initialises a complete copy of the original model. Flock initialises the migrated model by copying only those model elements from the original model that conform to the migrated metamodel. The effects of these different copying strategies can be seen in many of the examples in Table 6.1. To explain this, a smaller example of co-evolution is used below.

A Petri `Net` comprises `Places` and `Transitions` (Figure 6.1). A `Place` has any number of `src` or `dst Transitions`. Similarly, a `Transition` has at least one `src` and `dst Place`. The metamodel in Figure 6.1 is to be evolved so as to support weighted connections between `Places` and `Transitions` and between `Transitions` and `Places`.

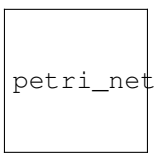

Figure 6.1: Original Petri nets metamodel.

The evolved metamodel is shown in Figure 6.2. `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`.

Figure 6.2: Evolved Petri nets metamodel.

Models that were consistent with the original metamodel may not be consistent with the evolved metamodel. For example, `Transition` objects can no longer define values for `src` and `dst` features, and must define at least one value for the `in` and `out` features.

The migration strategy for this evolution varies when specified in ATL, Groovy-for-COPE and Flock, largely due to the differences in the way in which each approach initialises migrated models. When specified in ATL, migration must copy values from original to migrated model, as shown on lines 5-6, 12 and 20 of Listing 6.1. When using Groovy-for-COPE, values contained in slots that no longer correspond to features in the migrated metamodel must be unset, as shown on lines 2, 9 and 18-19 of Listing 6.2. Finally, Flock initialises the migrated model by copying values only for those features that remain unchanged in the migrated model. Consequently, no explicit copying or unsetting is required in Listing 6.3; the migration strategy manipulates only metafeatures that do not exist in the evolved metamodel (`Transition#src` and `Transition#dst`) and the new metaclasses, `TPArc` and `PTArc`.

```
1  rule Nets {
2    from
3      o : Before!Net
4    to
5      m : After!Net ( places <- o.places, transitions <- o.
           transitions )
6  }
7
8  rule Places {
9    from
10     o : Before!Place
11   to
12     m : After!Place ( name <- o.name )
13 }
14
15 rule Transitions {
16   from
17     o : Before!Transition
18   to
19     m : After!Transition (
20       name <- o.name,
21       "in" <- o.src->collect(p | thisModule.PTArcs(p,o)),
22       out <- o.dst->collect(p | thisModule.TPArcs(o,p))
23     )
24 }
25
```

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

lazy rule TPArcs {
  from
    transition : Before!Transition,
    destination : Before!Place
  to
    tparcs : After!TPArc (
      src <- transition,
      dst <- destination,
      net <- transition.net
    )
}
```

Listing 6.1: Petri nets model migration in ATL

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

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

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

Listing 6.2: Petri nets model migration in COPE

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

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

```
15  }
```

Listing 6.3: Petri nets model migration in Flock

With regard to the way in which they initialise the migrated model, COPE and ATL are opposites. The former initialises an exact copy of the original model, and so unset operations must be used when the value of a feature should not have been copied. By contrast, the latter initialises an empty model, and so 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.

This difference explains the results shown in Table 6.1. Firstly, because Flock requires no unsetting of affected model elements nor explicit copying of unaffected model elements, less model operations are used when specifying a migration strategy with Flock than is used when specifying the same migration strategy with Groovy-for-COPE or with ATL. Secondly, there are more features unaffected by metamodel evolution than affected. Consequently, specifying model migration with ATL for the examples shown in Table 6.1 requires more model operations than in Groovy-for-COPE.

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 difference in copying strategy. Instead, the way in which models are initialised by the migration languages 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 Groovy-for-COPE and, when the feature is not renamed, also Flock) can have side-effects which complicate migration.

In the Change Reference to Containment example, a `System` initially comprises `Ports` and `Signatures` (Figure 6.3). A `Signature` references any number of `ports`. The metamodel is to be evolved so that `Ports` can no longer be shared between `Signatures`.

Figure 6.3: Original metamodel.

The evolved metamodel is shown in Figure 6.4. `Signatures` now contain - rather than reference - `Ports`. Consequently, the `ports` feature of `System` is no longer required and is removed.

Figure 6.4: Evolved metamodel.

The migration strategy is straightforward in ATL: 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`.

```
1  rule Systems {
2   from
3    o : Before!System
4   to
5    m : After!System ( signatures <- o.signatures )
6  }
7
8  rule Signature {
9   from
10   o : Before!Signature
11  to
12   m : After!Signature (
13    ports <- o.ports->collect(p | thisModule.Port(p))
14   )
15 }
16
17 lazy rule Port {
18  from
19   o : Before!Port
20  to
21   m : After!Port ( name <- o.name )
```

Listing 6.4: Change R to C model migration in ATL

In Flock and Groovy-for-COPE, migration is less straightforward because, during migration, 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 Groovy-for-COPE, the containment nature of the reference is not enforced until after the migration strategy is executed. Hence, the migration strategy 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, a Flock migration strategy requires one more more model operation than the equivalent ATL migration strategy: to delete model elements that are not contained in any instance of the containment reference (line 11 of Listing 6.6). A Groovy-for-COPE migration strategy requires three more model operations than the equivalent ATL migration strategy: one to delete model elements that are not contained in any instance of the containment reference (lines 17-21 of Listing 6.5), one to dereference model elements that have already been cloned (line 10 of Listing 6.5), and one to mark a model element as contained (line 12 of Listing 6.5). In the example considered here, the ATL migration strategy requires one more model operation than the Flock and Groovy-for-COPE migration strategies (to copy the contents of System#signature features from original to migrated model), due to the difference in copying strategy. This

leads to the result shown in Table 6.1: the Flock and ATL migration strategies have an equal number of model operations, while the COPE migration strategy has two more.

Summary

This section has compared the model migration languages

- Make clear the contribution: no other research has attempted to compare model migration languages. It's not clear how it should be done. This is a start. - Not trying to argue that the figures are statistically significant. Just an approximation of what we're trying to measure. - Extensions: measure cyclomatic complexity, etc

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

[Eclipse 2008b] Eclipse. Graphical Modelling Framework project [online]. [Accessed 19 September 2008] Available at: `http://www.eclipse.org/modeling/gmf/`, 2008.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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