A Comparison of Model Migration Tools

Louis M. Rose, Markus Herrmannsdoerfer, James R. Williams, Dimitris Kolovos, Kelly Garcés, Richard F. Paige, Fiona A.C. Polack

A Comparison of Model Migration Tools

Louis M. Rose[1], Markus Herrmannsdoerfer[2], James R. Williams[1], Dimitrios S. Kolovos[1], Kelly Garcés[3,4], Richard F. Paige[1], and Fiona A.C. Polack[1]

[1] Department of Computer Science,
University of York, UK.
{louis, jw, dkolovos, paige, fiona}@cs.york.ac.uk
[2] Institut für Informatik,
Technische Universität München, Germany.
herrmama@in.tum.de
[3] AtlanMod (EMN-INRIA)
Nantes, France.
[4] ASCOLA (LINA-INRIA)
Nantes, France.
kelly.garces@mines-nantes.fr

Abstract. Modelling languages and thus their metamodels are subject to change. When a metamodel evolves, existing models may no longer conform to the evolved metamodel. To avoid rebuilding them from scratch, existing models must be migrated to conform to the evolved metamodel. Manually migrating existing models is tedious and error-prone. To alleviate this, several tools have been proposed to build a migration strategy that automates the migration of existing models. Little is known about the advantages and disadvantages of the tools in different situations. In this paper, we thus compare a representative sample of migration tools – AML, COPE, Ecore2Ecore and Epsilon Flock – using common migration examples. The criteria used in the comparison aim to support users in selecting the most appropriate tool for their situation.

1 Introduction

When a metamodel evolves, existing models may no longer conform to the structures and rules of the metamodel [4]. To avoid rebuilding existing models from scratch, these models are migrated to conform to the evolved metamodel. Manual migration is tedious and error-prone, and so migration needs to be automated [11]. Building an automated migration strategy (even if desirable in practice) is non-trivial, as it has to correctly migrate an arbitrary set of models.

Recently, many different tools for building a migration strategy have become available. Each tool has strengths and weaknesses. However, little is known about how the tools compare in practice and so tool selection is difficult.

In this paper, we compare four model migration tools, selected from those described in Section 2. Following the systematic process outlined in Section 3, the tools are applied to two examples to facilitate their comparison. Section 4 reports our experiences in using each of the tools, highlighting their strengths

and weaknesses using nine criteria that we deem important for model migration. From this comparison, Section 5 synthesises advice and guidelines to help users in identifying the most appropriate model migration tool for their situation.

2 Related Work

Model transformation. Model migration can be implemented in a general-purpose programming language (such as Java), or in a model-to-model (M2M) transformation language, such as QVT [19] (the current OMG standard), ATL [15] or Xtend (of the popular openArchitectureWare framework[1]).

[17] identifies different kinds of model transformations, and in particular two categories of relationship between source and target metamodel: *exogenous* and *endogenous*. In the former, the source and target metamodels differ, and the target model is constructed entirely by the transformation. In the latter, source and target metamodels are the same, and so the target model can be initialised to be the same as source model before the transformation. In model migration, source and target metamodels differ, and hence endogenous transformations cannot be used. Consequently, model migration strategies are often specified with exogenous model-to-model transformation languages, and must contain sections for copying from original to migrated model those model elements that have not been affected by metamodel evolution.

Model migration. As was first argued by Sprinkle [22], model migration is best served by a language that combines properties of exogenous and endogenous model transformation: we need to be able to specify the transformation from a source metamodel to a different target metamodel, but only for the metamodel elements for which a migration is required. Rose et al. [20] classify model migration approaches into the following categories:

Manual specification approaches provide transformation languages to manually specify the model migration. These transformation languages try to reduce the effort for building a migration strategy by providing mechanisms that are specific for model migration. For instance, the approaches described in [18, 21, 23] extend an exogenous transformation language to automatically copy model elements whose metamodel definition has not changed. While manual specification fosters correctness of the model migration, it also requires the most effort to build a migration strategy.

Operator-based approaches, such as [12, 25], provide coupled operators that allow metamodel changes and model migration strategies to be specified together. By capturing recurring co-evolution patterns as operators, these approaches avoid the need to specify identity rules, reusing recurring combinations of metamodel evolution and model migration through coupled operations.

Metamodel matching approaches automatically generate an exogenous model transformation from the difference between two metamodel versions. Because

[1] `http://www.openarchitectureware.org/`

an exogenous transformation is generated, model elements that have not been affected by co-evolution must be considered. Unlike manual specification, boilerplate code for automatic copying is automatically generated. Cicchetti [1] was the first to report a metamodel matching approach, noting that some categories of change cannot be automatically migrated. Garcés et al. [7] provide a potentially more expressive approach that allows the matching strategy to be parameterised.

Comparison. Apart from the above categorisation based on theoretical aspects of existing model migration approaches, no work compares model migration tools. However, several papers compare model transformation languages. Czarnecki and Helsen [2] present a feature model to classify transformation languages according to their technical properties. Mens and van Gorp [17] present functional and non-functional requirements for transformation languages. Taentzer et al. [24] compare the graph transformation languages AGG, TGG, VIATRA, and VMTS using the well-known object to relational transformation example. Gronmo et al. [9] compare the transformation languages CGT, AGG, and ATL using a complex refactoring example. These comparisons are used here to derive criteria for the comparison of model migration tools.

3 Comparison Method

In this section, we present the approach used to compare the model migration tools. The comparison is based on practical application of the tools to the co-evolution examples presented in Section 3.1. The selection of tools for the comparison is described in Section 3.2. To contextualise the conclusions drawn in this paper, Section 3.3 describes the process used to carry out the comparison.

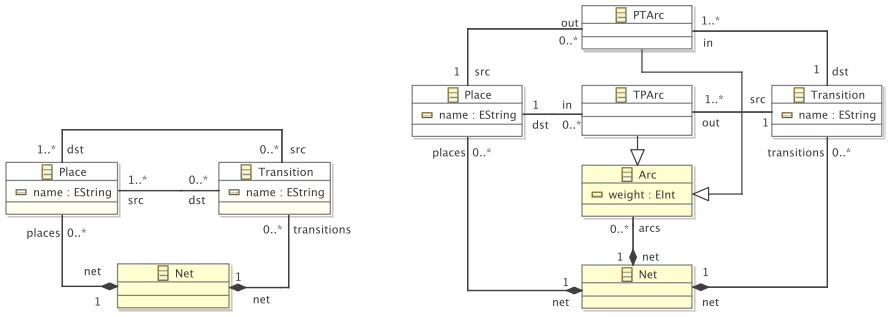
3.1 Co-Evolution Examples

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 comparison process, as discussed in Section 3.3. 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 [13].

Petri Nets. The first example is an evolution of a Petri net metamodel, previously used in [1, 7, 21, 25] to discuss co-evolution and model migration.

In Figure 1(a), a Petri `Net` comprises `Places` and `Transitions`. A `Place` has any number of `src` or `dst` `Transitions`. Similarly, a `Transition` has at least one `src` and `dst` `Place`. In this example, the metamodel in Figure 1(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 1(b). `Places` are connected to `Transitions` via instances of `PTArc`. Likewise, `Transitions` are connected to `Places` via `TPArc`. Both `PTArc` and `TPArc` inherit from `Arc`, and therefore can be used to specify a `weight`.



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

Fig. 1. Petri nets metamodel evolution (taken from [21]). Shading is irrelevant.

GMF. The second example is taken from the Graphical Modeling Framework (GMF) [8], 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[2], 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 [5] 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`).

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

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

editors provided by GMF were used to determine the correctness of the migration strategies produced by using each model migration tool.

3.2 Compared Tools

For the comparison in this paper, we selected one tool from each of the three categories – *manual specification*, *operator-based* and *metamodel matching* approaches – described in Section 2. We included a further tool from the manual specification category, Ecore2Ecore, as it is distributed with the Eclipse Modeling Framework, arguably the most widely used modelling framework. Each of these tools is discussed briefly below. Section 4 describes each tool in more detail.

AtlanMod Matching Language (AML) [7, 6] is a model matching tool, which can be used as a *metamodel matching* migration tool. AML provides heuristics that the user combines to specify a metamodel matching strategy. A migrating ATL transformation is automatically generated by matching original and evolved metamodels.

COPE [12] is an *operator-based* migration tool. COPE provides a library of *co-evolutionary operators*. Each co-evolutionary operator specifies both a metamodel evolution and a corresponding model migration strategy. For example, the "Introduce Reference Class" operator from COPE evolves the metamodel such that a reference is replaced by a class and migrates models such that links conforming to the reference are replaced by instances of the reference class.

Ecore2Ecore [14] is a *manual specification* migration tool that is part of the Eclipse Modeling Framework (EMF). Migration is specified with a mapping model and hand-written Java code. Ecore2Ecore has been used in real-world projects, such as the Eclipse MDT UML2 project [3], to manage co-evolution.

Epsilon Flock [21] (subsequently referred to as Flock) is a *manual specification* migration tool. Flock is a domain-specific transformation language tailored for model migration. In particular, Flock automatically copies from original to migrated model all model elements that have not been affected by metamodel evolution. Flock is built atop Epsilon[3] [16], an extensible platform providing inter-operable programming languages for model-driven development.

3.3 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.

We began the comparison by allocating responsibility for using each tool on the examples to a different person. Because the authors of Ecore2Ecore and AML were not able to participate fully in the comparison, two colleagues experienced

[3] `http://www.eclipse.org/gmt/epsilon`

in model transformation and migration 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.

Table 1. 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

The comparison was conducted in three phases. In the first phase, we identified criteria against which the tools would be compared. In the second phase, we used the first example of co-evolution (Petri nets) to familiarise ourselves 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 conclusions were drawn from our experiences. Table 1 summarises the comparison criteria used in this paper. Further criteria could develop as a result of further experimentation in the future. The next section presents, for each criterion, observations from applying the migration tools to the co-evolution examples.

4 Comparison Results

By applying the method described in Section 3, four model migration tools were compared. This section reports similarities and differences of each tool, using nine criteria. Each subsection considers one criterion. The complete solutions are available online[4].

4.1 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

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

to produce an executable workflow. The workflow is invoked to generate the migrating transformation, codified in the Atlas Transformation Language (ATL) [15]. 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 [7, 6]) 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. 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.

4.2 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.

4.3 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].

4.4 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.

4.5 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.

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

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.

4.6 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.

4.7 Expressiveness

Migration strategies are easier to infer for some categories of metamodel change than others [10]. 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

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

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 [10].

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.

4.8 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 4.1. 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.

Flock depends on Epsilon and its development tools also require Eclipse. Flock assumes that the original and target metamodels are available for encoding

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

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.

4.9 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.

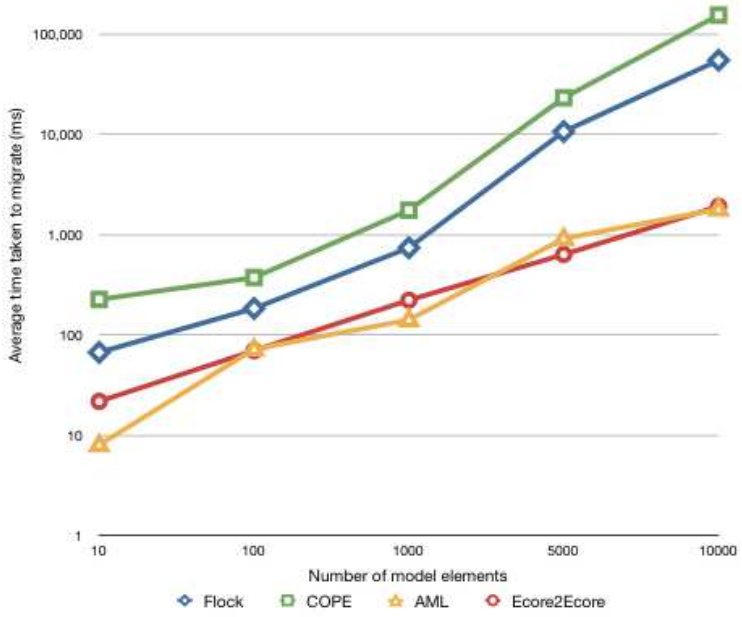


Fig. 2. Migration tool performance comparison.

To measure performance, we produced Petri net models with a random generator, varying their size. Figure 2 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 2 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.

5 Discussion and Conclusions

The comparison results highlight the similarities and differences between a representative sample of model migration approaches. In this section, the differences are used to consider which tools are better suited to particular model migration situations.

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 [11] 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.

Conclusions This paper has compared a representative sample of approaches to automating model migration, an activity crucial for supporting software evolution in MDE. The comparison was performed by following a methodical process and used 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 2.

The criteria considered in this paper provide a foundation for further comparisons. For example, we recognise the importance of the usability and learnability of migration tools, and envisage a comprehensive user study (with 100s of users) for assessing these criteria. Future work will identify further comparison criteria and conduct further experimentation. In particular, we plan to investigate memory usage and forward-compatibility of tools.

Table 2. 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

Acknowledgement. The work in this paper was supported by the European Commission via the MADES project, co-funded under the "Information Society Technologies" 7th Framework Programme (2009-2012). The work of the second author was funded by the German Federal Ministry of Education and Research (BMBF), grants "SPES2020, 01IS08045A" and "Quamoco, 01IS08023B". The work of the third author was supported by the EPSRC, through the Large-Scale Complex IT Systems project, "EP/F001096/1". The authors thank Kenn Hussey and Marcelo Paternostro for reviewing a draft of this paper.

References

1. A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Automating co-evolution in MDE. In *Proc. EDOC*, pages 222–231. IEEE Computer Society, 2008.
2. K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645, 2006.
3. Eclipse. UML2 Model Development Tools project [online]. [Accessed 7 September 2009] Available at: `http://www.eclipse.org/modeling/mdt/uml2`, 2009.
4. J. Favre. Meta-model and model co-evolution within the 3d software space. In *Proc. ELISA Workshop*, pages 98–109, September 2003.

5. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.
6. K. Garcés, F. Jouault, P. Cointe, and J. Bézivin. A Domain Specific Language for Expressing Model Matching. In *Proc. IDM*, Nancy, France, 2009.
7. K. Garcés, F. Jouault, P. Cointe, and J. Bézivin. Managing model adaptation by precise detection of metamodel changes. In *Proc. ECMDA-FA*, volume 5562 of *LNCS*, pages 34–49. Springer, 2009.
8. R.C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 2009.
9. R. Grønmo, B. Møller-Pedersen, and G.K. Olsen. Comparison of three model transformation languages. In *Proc. ECMDA-FA*, volume 5562 of *LNCS*, pages 2–17. Springer, 2009.
10. B. Gruschko, D.S. Kolovos, and R.F. Paige. Towards synchronizing models with evolving metamodels. In *Workshop on Model-Driven Software Evolution*, 2007.
11. M. Herrmannsdoerfer, S. Benz, and E. Juergens. Automatability of coupled evolution of metamodels and models in practice. In *Proc. MoDELS*, volume 5301 of *LNCS*, pages 645–659. Springer, 2008.
12. M. Herrmannsdoerfer, S. Benz, and E. Juergens. COPE - automating coupled evolution of metamodels and models. In *Proc. ECOOP*, volume 5653 of *LNCS*, pages 52–76. Springer, 2009.
13. M. Herrmannsdoerfer, D. Ratiu, and G. Wachsmuth. Language evolution in practice. In *Proc. SLE*, volume 5696 of *LNCS*, pages 3–22. Springer, 2009.
14. K. Hussey and M. Paternostro. Advanced features of EMF. Tutorial at EclipseCon 2006, California, USA. [Accessed 07 September 2009] Available at: `http://www.eclipsecon.org/2006/Sub.do?id=171`, 2006.
15. F. Jouault and I. Kurtev. Transforming models with ATL. In *Proc. Satellite Events at MoDELS*, volume 3844 of *LNCS*, pages 128–138. Springer, 2005.
16. D.S. Kolovos. *An Extensible Platform for Specification of Integrated Languages for Model Management*. PhD thesis, University of York, United Kingdom, 2009.
17. T. Mens and P. Van Gorp. A taxonomy of model transformation. *Electron. Notes Theor. Comput. Sci.*, 152:125–142, 2006.
18. A. Narayanan, T. Levendovszky, D. Balasubramanian, and G. Karsai. Automatic domain model migration to manage metamodel evolution. In *Proc. MoDELS*, volume 5795 of *LNCS*, pages 706–711. Springer, 2009.
19. OMG. Query/View/Transformation 1.0 Specification [online]. [Accessed 26 April 2010] Available at: `http://www.omg.org/spec/QVT/1.0/`, 2008.
20. L.M. Rose, D.S. Kolovos, R.F. Paige, and F.A.C. Polack. An analysis of approaches to model migration. In *Proc. Joint MoDSE-MCCM Workshop*, 2009.
21. L.M. Rose, D.S. Kolovos, R.F. Paige, and F.A.C. Polack. Model migration with Epsilon Flock. In *Proc. ICMT [accepted and to appear]*, 2010.
22. J. Sprinkle. *Metamodel Driven Model Migration*. PhD thesis, Vanderbilt University, TN, USA, 2003.
23. J. Sprinkle, A. Agrawal, T. Levendovszky, F. Shi, and G. Karsai. Domain model evolution in visual languages using graph transformations. In *Proc. Workshop on Domain-Specific Visual Languages*, 2002.
24. Gabriele Taentzer, Karsten Ehrig, Esther Guerra, Juan De Lara, Tihamer Levendovszky, Ulrike Prange, and Daniel Varro. Model transformations by graph transformations: A comparative study. In *Model Transformations in Practice Workshop at MoDELS 2005, Montego*, page 05, 2005.
25. G. Wachsmuth. Metamodel adaptation and model co-adaptation. In *Proc. ECOOP*, volume 4609 of *LNCS*, pages 600–624. Springer, 2007.