Chapter 6

Evaluation

This chapter seeks to determine the extent to which the structures and processes proposed in this thesis are beneficial in terms of increased productivity and understandability of software development. The co-evolution process identified in Chapter 4 and the dedicated structures for managing co-evolution, which were described in Chapter 5, are evaluated by comparison to other structures, application to real-world examples, and expert evaluation. The evaluation presented here used the co-evolution examples described in Appendices B and C, which are distinct from the co-evolution examples that were used for analysis in Chapter 4.

Chapter 4 identified *user-driven co-evolution*, a process for managing co-evolution that had been used in real-world MDE projects, but had not been recognised in the literature. Chapter 5 described the implementation of two structures tailored for user-driven co-evolution, a *metamodel-independent syntax* and a *textual modelling notation*. Using a real-world example of user-driven co-evolution, Section 6.1 assesses the extent to which the dedicated structures proposed in Chapter 5 affect the productivity of user-driven co-evolution.

The remainder of the chapter focuses on developer-driven co-evolution (in which a migration strategy is specified in an executable format) and on *Epsilon Flock* (Section 5.4), a transformation language tailored for model migration. Section 6.2 evaluates the novel source-target relationship strategy implemented in Flock, *conservative copy*, by comparison to two existing source-target relationship strategies using co-evolution examples from real-world projects.

While Section 6.2 focuses on assessing conservative copy, Sections 6.3 and 6.4 evaluate Flock as a whole, using an expert evaluation and a transformation contest, respectively. In both sections, evaluation is performed using large, independent examples of co-evolution.

The work presented in this chapter has been published in [Rose *et al.* 2010a, Rose *et al.* 2010d, Rose *et al.* 2010c]. The evaluation described in Sections 6.3

and 6.4 was performed collaboratively, and the contributions of others are highlighted in those sections.

6.1 Evaluating User-Driven Co-Evolution

This section evaluates the suitability of the metamodel independent syntax (Section 5.1) and the textual modelling notation (Section 5.2) for performing user-driven co-evolution, a novel process for managing co-evolution that was identified in Chapter 4. User-driven co-evolution was observed in real-world MDE projects in Chapter 4, but no tool support for user-driven co-evolution is reported in the co-evolution literature. The evaluation presented in this section seeks to demonstrate that dedicated structures for user-driven co-evolution can increase the productivity of model migration in a user-driven co-evolution process.

To explore this claim, several approaches to evaluation could have been used. The dedicated structures for user-driven co-evolution are freely available as part of Epsilon, a member of the Eclipse Modeling Project. The productivity benefits of the structures might have been explored by asking users to describe their experiences with the structures. However, the subjective nature of the feedback might have threatened the validity of this evaluation. Alternatively, evaluation might have been performed with a comprehensive user study that measured the time taken for developers to perform model migration with and without the dedicated structures for user-driven co-evolution. However, locating developers and examples of co-evolution for this study was not possible given the time available to perform the evaluation.

Instead, evaluation was conducted by comparing two approaches to user-driven co-evolution using an example of user-driven co-evolution from a real-world MDE project. The first approach uses only those tools available in EMF; while the second approach uses EMF and, in addition, two of the structures presented in Chapter 5, a metamodel-independent syntax and a textual modelling notation. The remainder of this section first recaps Section 4.2.2, which describes the challenges to productivity faced by developers while performing user-driven co-evolution with the Eclipse Modeling Framework (EMF), arguably the most widely-used MDE development environment at present. Subsequently, the two approaches to user-driven co-evolution are demonstrated. The section concludes by comparing the two approaches and highlighting ways in which two of the structures proposed in Chapter 5 might be used to increase the productivity of model migration in a user-driven co-evolution process.

6.1.1 Challenges for Performing User-Driven Co-Evolution

Chapter 4 highlighted two productivity challenges faced by developers while performing user-driven co-evolution in contemporary MDE environments. 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, the multi-pass parsers used to load models in contemporary MDE environments cause user-driven co-evolution to be an iterative process, because not all conformance errors are reported at once. The identification of these productivity 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.*

Two of the structures presented in Chapter 5 provide the foundation for fulfilling the above research requirement. The first, a metamodel-independent syntax, facilitates the conformance checking of a model against any metamodel. The second structure, a textual modelling notation called HUTN, allows models to be managed in a format that is reputedly easier for humans to use than XMI, the canonical model storage format [OMG 2004].

To fulfil the above research requirement, this section uses the metamodel-independent syntax and the textual modelling notation to demonstrate that user-driven co-evolution can be performed without encountering the challenges to productivity described above. To this end, an example of co-evolution is used to show the way in which user-driven co-evolution might be achieved with and without the metamodel-independent syntax and the textual modelling notation described in Chapter 5.

6.1.2 Co-Evolution Example

The remainder of this section uses a co-evolution example taken from collaborative work with Adam Sampson, then a Research Associate at the University of Kent. The purpose of the collaboration was to build a prototypical editor for graphical models of programs written in process-oriented programming languages, such as occam-π [Welch & Barnes 2005]. The graphical models would provide a standard notation for describing process-oriented programs.

The graphical model editor was developed using MDE tools and techniques. A metamodel was used to capture the abstract syntax of process-oriented programming languages, and code for a graphical model editor was automatically generated from the metamodel.

The final version of the graphical model editor is shown in Figure 6.1. The editor captures the three primary concepts used to specify process-oriented programs: processes, connection points and channels. Processes, represented as boxes in the graphical notation, are the fundamental building blocks of a process-oriented program. Channels, represented as lines in the graphical notation, are the mechanism by which processes communicate, and are unidirectional. Connection points, represented as circles in the graphical notation, define the channels on which a process can communicate. Connection points are used to specify the way in which a process can communicate, and can optionally be bound to a channel. Because channels are unidirectional, connection points are either reading (consume messages from the channel) or writing (generate messages on the channel). Reading (writing) connection points are represented as white (black) circles in the graphical notation.

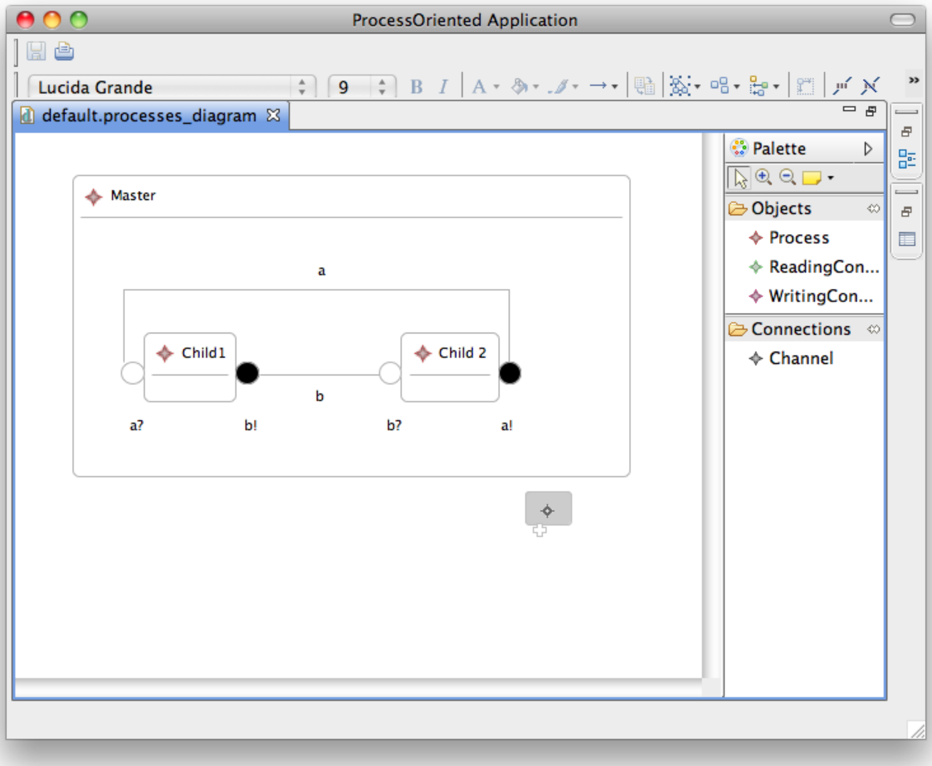


Figure 6.1: Final version of the prototypical graphical model editor.

The graphical model editor was implemented using EMF. The metamodel was specified in Ecore, the metamodelling language of EMF, and the editor was generated from the metamodel using the Graphical Modeling Framework (GMF), an extension to EMF for graphical modelling. Section 2.3 describes in more detail the way in which EMF and GMF can be used to specify metamodels and to generate graphical model editors.

The process-oriented metamodel was developed iteratively, and the six iterations are described in Appendix B. During each iteration, the metamodel was changed. The remainder of this section uses an example of metamodel changes from the fifth iteration of the project. The way in which development proceeded during that iteration is described in Section B.5 and summarised below.

Feature identification

The purpose of the iteration was to refine the way in which connection points were represented. At the start of the iteration, the graphical model editor could be used to draw processes, channels and connection points. However, no distinction was made between reading and writing connection points.

Figure 6.2 shows an exemplar model represented in the graphical model editor before the iteration began. The model contains two processes (depicted as boxes), `P1` and `P2`, one channel (depicted as a line), `a`, and two connection points (depicted as circles), `a!` and `a?`.

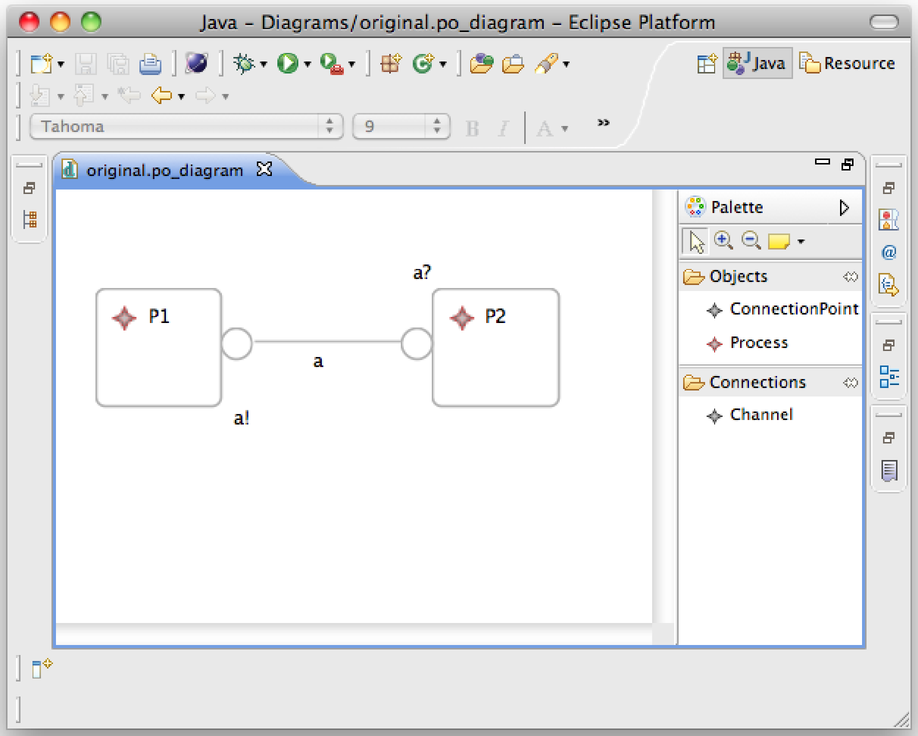


Figure 6.2: The graphical editor at the start of the iteration.

The aim of the iteration was to distinguish between *reading* and *writing* connection points in the graphical notation. The former are used to receive messages, and the latter to send messages. In Figure 6.2, `a?` is intended to represent a reading connection point, and `a!` a writing connection point. Sampson and the thesis author decided that the editor should be changed so that black circles would be used to represent writing connection points, and white circles to represent reading connection points. At the end of the iteration the model shown in Figure 6.2 would be represented as shown in Figure 6.3. Furthermore, the editor would ensure that `a?` was used only as the reader of a channel, and `a!` only as the writer of a channel. Before the iteration started, the editor did not enforce this constraint.

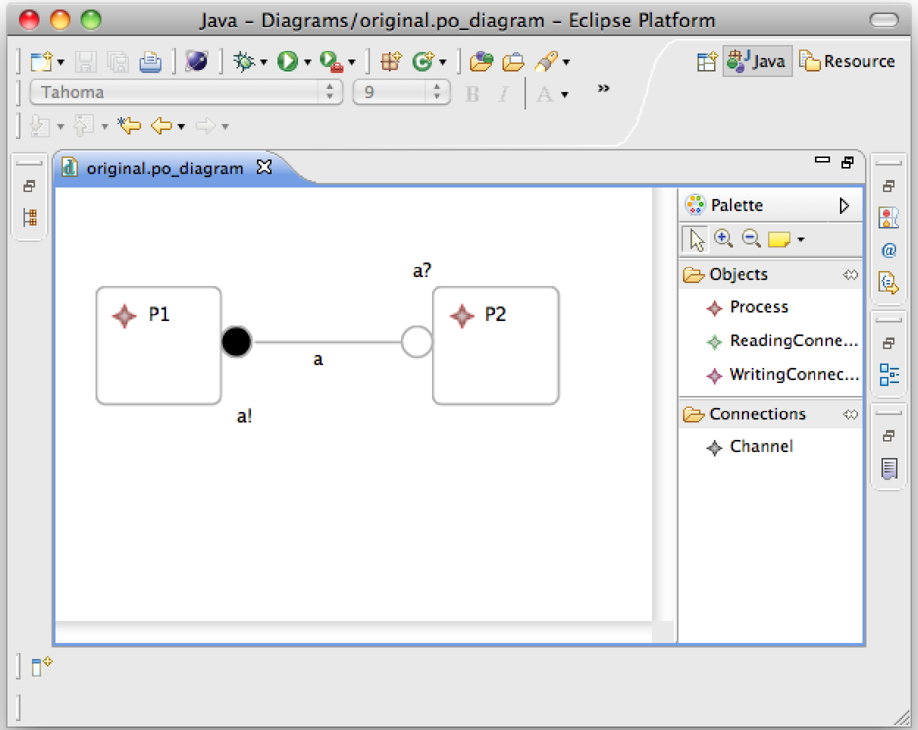


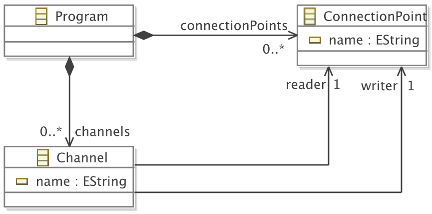
Figure 6.3: The graphical editor at the end of the iteration.

Implementation

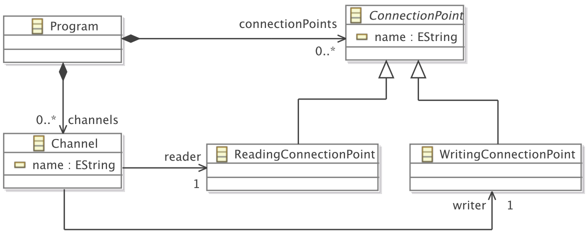
Before the iteration began, the metamodel did not distinguish between reading and writing `ConnectionPoints`. Figure 6.4(a) shows the way in which connection points were modelled at the start of the iteration. When a `ConnectionPoint` was associated with a `Channel`, the `ConnectionPoint` is specified as a `reader` or a `writer` for that `Channel`, and otherwise the type of a `ConnectionPoint` is not specified.

The way in which connection points were modelled was changed, resulting in the metaclasses shown in Figure 6.4(b). `ConnectionPoint` was made abstract, and two subtypes, `ReadingConnectionPoint` and `WritingCo-`

`nnectionPoint`, were introduced. The `reader` and `writer` references of `Channel` were changed to refer to the new subtypes.



(a) Part of the original metamodel.



(b) Part of the evolved metamodel.

Figure 6.4: Process-oriented metamodel evolution.

Following the metamodel changes, a new version of the graphical editor was generated automatically from the metamodel using GMF. An annotation – not shown in Figure 6.4(b) – on the `WritingConnectionPoints` was used to indicate to GMF that black circles were to be used to represent writing connection points in the graphical notation.

Testing

Testing the new version of the graphical editor highlighted the need for model migration. Attempting to load existing models, such as the one shown in Figure 6.2, caused an error because `ConnectionPoint` was now an abstract class. Any model specifying at least one connection point no longer conformed to the metamodel. Model migration was performed to re-establish conformance and to allow the models to be loaded.

Several models, presented in Appendix B had been constructed when testing previous versions of the graphical editor. The models were used during each iteration to ensure that any changes had not introduced regressions. Following the metamodel changes described above, the models could no longer be loaded and required migration. A user-driven rather than a developer-driven co-evolution approach was preferred throughout the development of process-oriented editor because only a few small models required migration in each iteration.

The remainder of this section describes the way in which model migration was performed for the changes to the process-oriented metamodel outlined above. The sequel describes the way in which migration was performed during the development of the process-oriented metamodel, without dedicated structures for performing user-driven co-evolution. Section 6.1.4 describes the way in which migration could have been performed using two of the structures presented in Chapter 5. The section concludes by comparing the two approaches.

6.1.3 User-Driven Co-Evolution with EMF

During the development of the process-oriented metamodel, no structures for performing user-driven co-evolution were available. Instead, migration was performed using only those tools available in EMF, as described below.

Migration with EMF involved identifying and fixing conformance errors, using the approach shown in Figure 6.5. When a model is loaded by the graphical editor, EMF automatically reports any conformance problems. Because EMF cannot load non-conformant models, the underlying storage representation of the model, XMI, is edited by hand to reconcile conformance problems. After saving the reconciled XMI to disk, the model is loaded again in the graphical editor and, because EMF uses a multi-pass XMI parser, additional conformance problems might be reported. If further conformance problems are reported, additional changes are made to the XMI. Otherwise, migration is complete.

EMF reports conformance problems when a model is loaded. For the process-oriented metamodel, loading a model involved opening the model with the graphical model editor. For the model shown in Figure 6.2, the conformance problems shown in the bottom pane (and by the error markers in the left-hand margin of the top pane) of Figure 6.6 were reported by EMF. For example, the first conformance problem reported is shown in the tooltip in Figure 6.2, and states that a `ClassNotFoundException` was encountered because the "Class 'ConnectionPoint' is not found or is abstract."

The conformance problems were fixed by editing the XMI shown in Figure 6.6, changing the type of the connection point objects to either a reading or a writing connection point. A reading (writing) connection point was used when the connection point was referenced via the `reader` (`writer`) reference of `Channel`. The reconciled XMI is shown in Figure 6.7. On lines 4 and 7, the connection point model elements have been changed to include `xmi:type` attributes, which specify whether the connection point should instantiate `ReadingConnectionPoint` or `WritingConnectionPoint`.

Reconciling the conformance problems by editing the XMI required considerable knowledge of the XMI specification. For example, the `xmi:type` attribute is used to specify the type of the connection point model elements. In fact, it must be included for those model elements. However, for the other

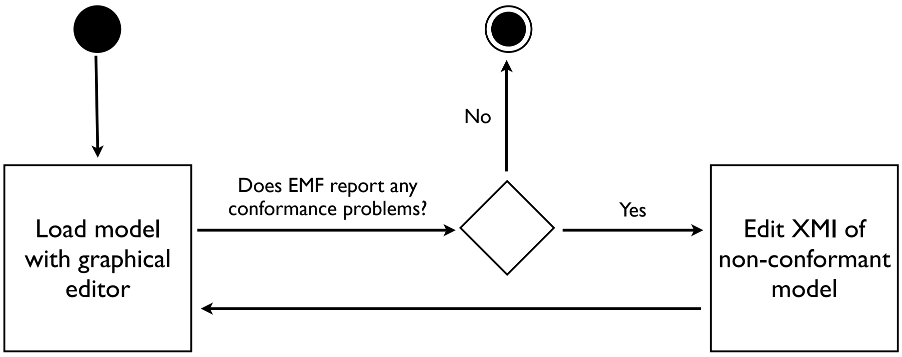


Figure 6.5: User-driven co-evolution with EMF

model elements in Figure 6.7 the `xmi:type` attribute is not necessary, and is omitted. When and how to use the `xmi:type` attribute is discussed further in the sidebar, and is synthesised from the XMI specification [OMG 2007c] and [Steinberg *et al.* 2008]. EMF abstracts away from XMI, and typically users do not interact directly with XMI. Therefore, it may be reasonable to assume that EMF users might not be familiar with XMI, and implementation details such as the `xmi:type` attribute.

The `xmi:type` attribute

In XMI, each model element specifies a value for the `xmi:type` attribute to indicate the metaclass that the model element instantiates. For example, the model element definition on line 4 of Figure 6.7 instantiates the metaclass named `WritingConnectionPoint`. To reduce the size of models on disk, the XMI specification allows type information to be omitted when it can be inferred. For example, line 9 of Figure 6.7 defines a model element that is contained in the `channels` reference of a `Process`. Because the `channels` reference can contain only one type of model element (`Channel`), the `xmi:type` attribute can be omitted, and the type information is inferred from the metamodel.

During the development of the process-oriented editor, some mistakes were made when migrating Sampson's models. For example, the wrong subtype of `ConnectionPoint` was used as the type of several connection point model elements. The mistake occurred because XMI identifies model elements using an offset from the root of the document. For example, consider the XMI shown in Figure 6.7. The channel on line 9 specifies the value "//@processes.1/@connectionPoints.0" for its `reader` attribute. The value is an XMI path referencing the first connection point ("@connectionPoints.0") contained in the second process ("@processes.1") of this document ("//");

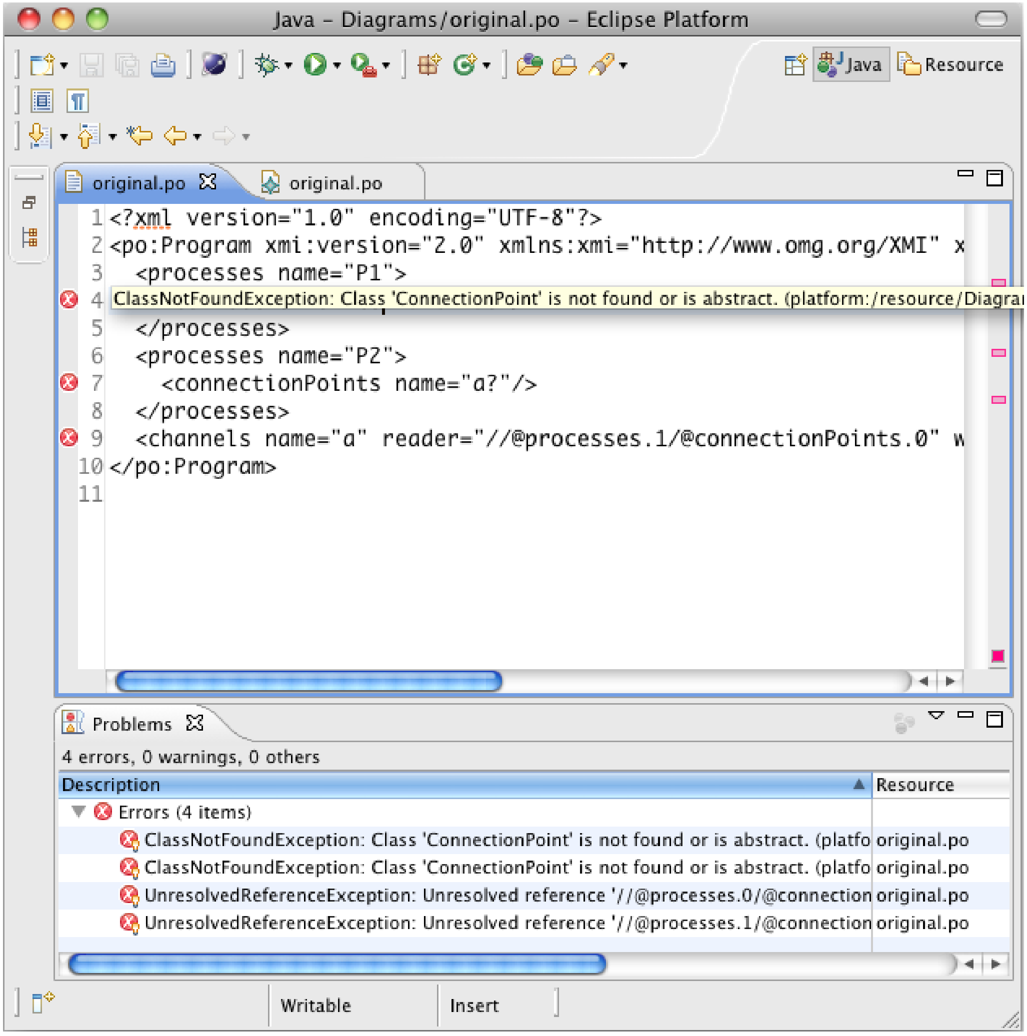


Figure 6.6: XMI prior to migration

in other words the connection point on line 7. One of Sampson's models contained many channels and connection points and incorrectly counting the connection points in the model led to several mistakes during the manual editing of the XMI.

As demonstrated above, migration using only the tools provided by EMF was an iterative process, and mistakes were made. The sequel demonstrates that, using the dedicated structures described in Chapter 5, migration can be performed in one iteration, without requiring the developer to switch between a conformance reporting and reconciling tools. In addition, the sequel suggests how the mistake described above might be avoided by using a textual modelling notation that has been optimised for human-use.

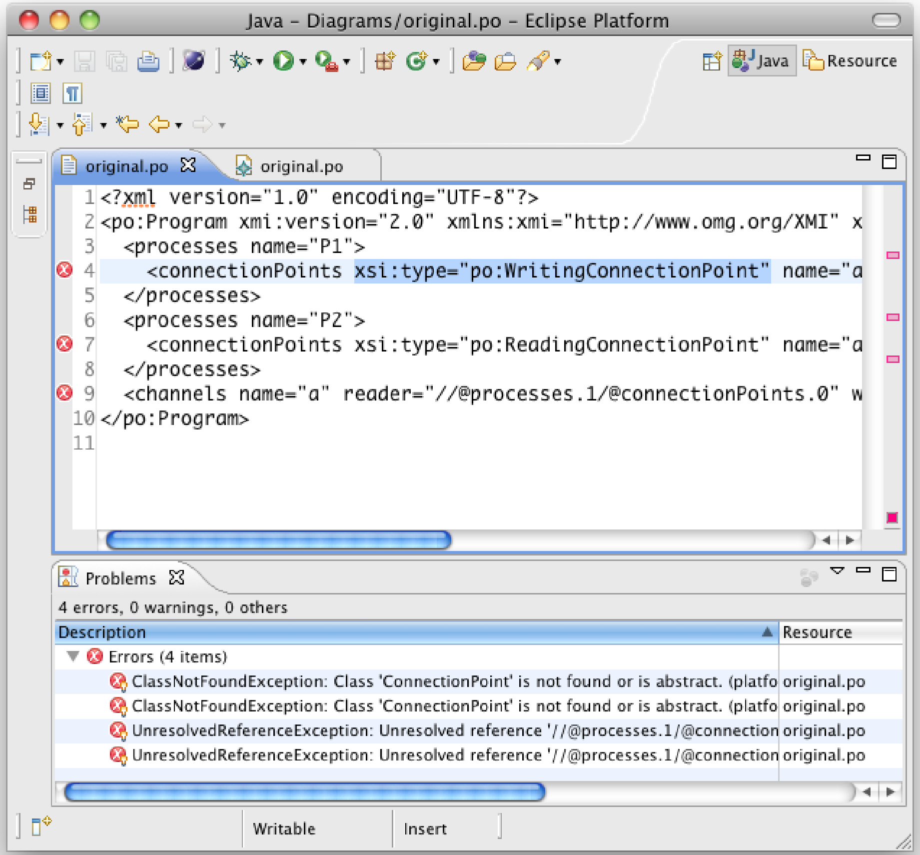


Figure 6.7: XMI after migration

6.1.4 User-Driven Co-Evolution with Dedicated Structures

Chapter 5 describes two structures that can be used to perform user-driven co-evolution. Here, the functionality of the two structures, a metamodel-independent syntax and a textual modelling notation, is summarised. Subsequently, an approach that uses the metamodel-independent syntax and the textual modelling notation for migrating the model from the process-oriented example is presented. The model migration example presented in this section was performed after the process-oriented editor was completed, and demonstrates how migration might have been achieved with dedicated structures for user-driven co-evolution. The sequel compares the user-driven co-evolution approach presented in this section with the approach presented above.

The metamodel-independent syntax presented in Section 5.1 allows non-conformant models to be loaded, and for the conformance of models to be checked against any metamodel. The textual modelling notation presented in Section 5.2 is built atop the metamodel-independent syntax and provides an alternative to XMI for editing models with a textual representation. Together, the two structures can be used for performing user-driven co-evolution using the approach shown in Figure 6.8. First, the user attempts to load a model. When the model cannot be loaded, the user clicks the "Generate HUTN" menu item, and the model's XMI is automatically transformed to HUTN by the implementation of HUTN described in Chapter 5. The generated HUTN source codes is presented in an editor that automatically reports conformance problems. The user edits the HUTN to reconcile conformance problems. The conformance report is automatically updated while the user edits the HUTN source. When the conformance problems are fixed, XMI for the conformant model is automatically generated, and migration is complete. The model can then be loaded in the graphical editor.

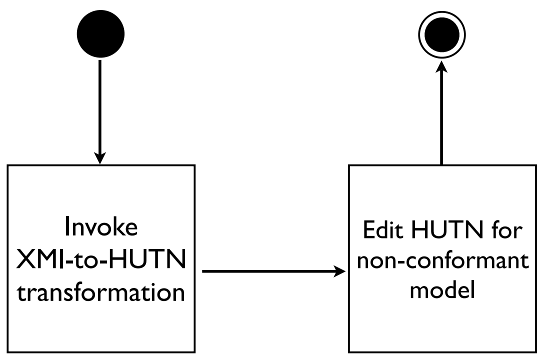


Figure 6.8: User-driven co-evolution with dedicated structures

The way in which the approach shown in Figure 6.8 might have been used for performing user-driven co-evolution for the process-oriented metamodel is now demonstrated. For the model shown in Figure 6.2, the HUTN shown in Figure 6.9 was generated by invoking the automatic XMI-to-HUTN transformation. The HUTN development tools automatically present any conformance problems, as shown in the bottom pane (and the left-hand margin of the top pane) in Figure 6.9.

Conformance problems are reconciled manually by the user, who edits the HUTN source. Conformance is automatically checked whenever the HUTN is changed. For example, Figure 6.10 shows the HUTN editor when migration is partially complete. Some of the conformance problems have been reconciled, and the associated error-markers are no longer displayed in the left-hand margin.

As discussed in Section 6.1.3, mistakes were made when migrating one of Sampson's models, probably because of the way in which XMI specifies refer-

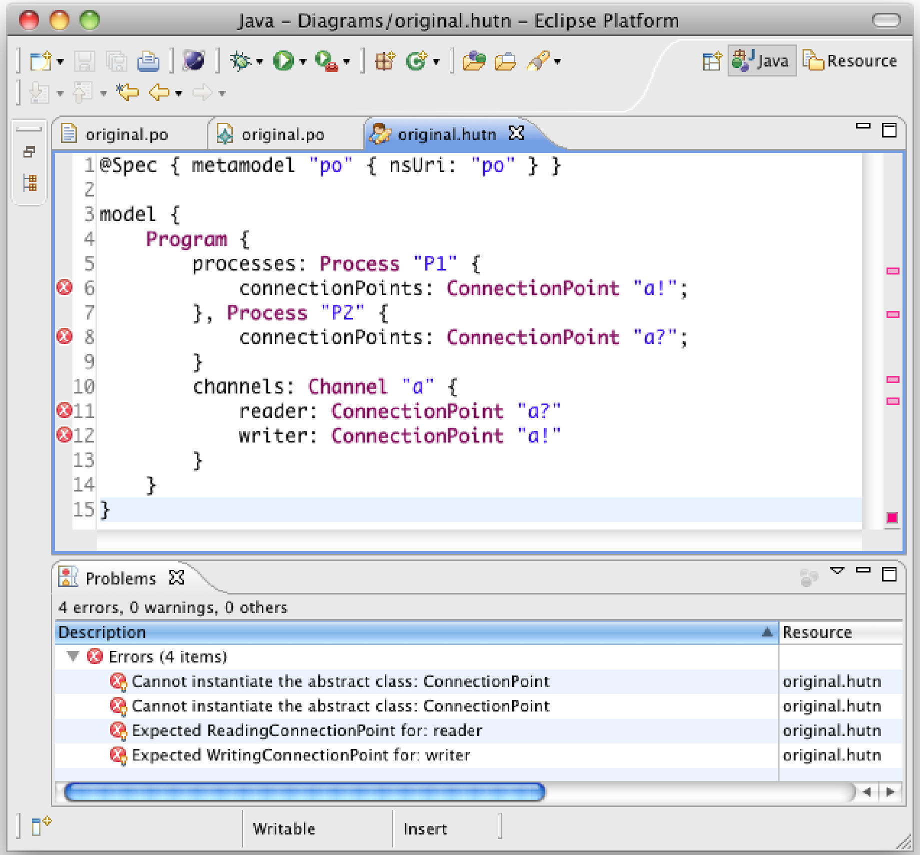


Figure 6.9: HUTN source prior to migration

ence values. In HUTN, reference values are specified by name. The channel on lines 10-13 of Figure 6.9 refers to its reader and writer by name ("a?" and "a!" respectively), rather than using a path (such as "//@processes.1/@connectionPoints.0") as is the case with XMI.

6.1.5 Comparison

To suggest ways in which dedicated structures for user-driven co-evolution might increase developer productivity, the two user-driven co-evolution approaches demonstrated above are now compared. The first approach, described in Section 6.1.3, uses only those tools available in EMF for performing user-driven co-evolution, while the second approach, described in Section 6.1.4 uses two of the structures described in Chapter 5. Applying the approaches to the process-oriented example highlighted differences between the modelling

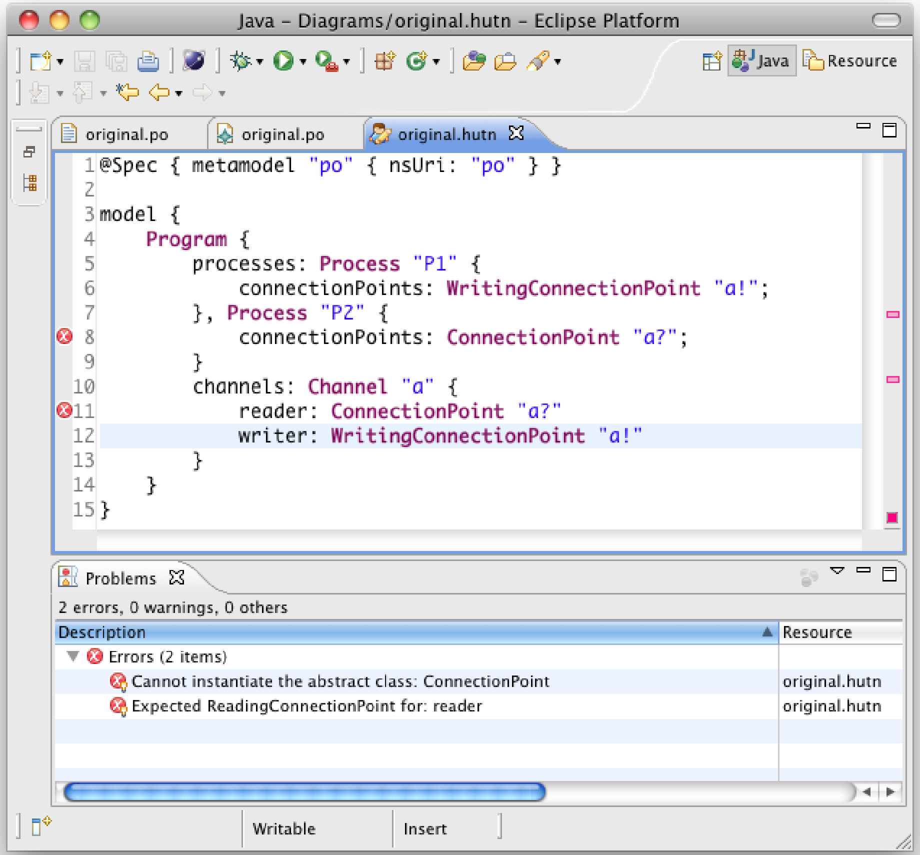


Figure 6.10: HUTN source part way through migration

notations used, and the way in which conformance problems were reported.

Differences in modelling notation

The two approaches used different modelling notations for reconciling conformance problems. The first approach used XMI, while the second used HUTN. The differences in notation that might influence developer productivity during user-driven co-evolution are now discussed. However, further work is required to more rigorously explore the extent to which developer productivity is affected by the modelling notation, as discussed in Section 6.1.6.

The way in which the type of a model element is specified varies between XMI and HUTN. In XMI, type information can be omitted in some circumstances, but must be included in others. In HUTN, type information is mandatory for every model element. Consequently, every HUTN document contains examples of how type information should be specified, whereas XMI documents may not.

Reference values are specified using paths in XMI (such as "//@processes.1/@connectionPoints.0") and by name (such as "a?") in HUTN. XMI paths are constructed in terms of a document's structure and, as such, rely on implementation details. The name of a model element, on the other hand, is specified in the model, and does not rely on any implementation details. Consequently, it is conceivable that fewer mistakes will be made during user-driven co-evolution when reference values are specified by name rather than using an XMI path.

Differences in conformance reports

The two approaches varied in the way in which conformance problems were reported, and, as a consequence, the first approach was iterative and the second was not. The way in which these differences might influence developer productivity during user-driven co-evolution are now discussed. Again, further work is required to more rigorously explore the extent to which developer productivity is affected by the differences in conformance reporting, as discussed in Section 6.1.6.

With EMF, user-driven co-evolution is an iterative process. Conformance errors are fixed by the user, who then reloads the reconciled model (with, for example, a graphical editor). Each time the model is loaded, further conformance problems might be reported when, for example, the user makes a mistake when reconciling the model. By contrast, the implementation of HUTN described in Section 5.2 uses a background compiler that checks conformance while the user edits the HUTN source. When the user makes a mistake reconciling the HUTN source, the error is reported immediately, and does not require the model to be loaded in the graphical editor.

Although not demonstrated in the example considered in this section, user-driven co-evolution would, for some types of metamodel changes, remain an iterative process even if EMF performed conformance checking in the background. Because EMF uses a multi-pass parser, some types of conformance problem are reported before other types. For example, conformance problems relating to multiplicity constraints (for example, a process does not specify a name, but name is a mandatory attribute) are reported after all other types of conformance problem. When several types of conformance problem have been affected by metamodel changes, user-driven co-evolution with EMF would remain an iterative process. Single-pass, background parsing is required to display all conformance problems while the user migrates a model.

6.1.6 Towards a more thorough comparison

Although the above comparison suggests that dedicated structures for performing user-driven co-evolution might increase developer productivity, further research is required to more rigorously evaluate this claim. The ways in which this evaluation might be extended in the future are now discussed.

A comprehensive user study, involving hundreds of users, is one means for exploring the extent to which productivity varies when dedicated structures are used to perform user-driven co-evolution. Ideally, participants for the study would constitute a large and representative sample of the users of EMF. Productivity might be measured by the time taken to perform co-evolution. To remove a potential source of bias, several examples of co-evolution might be used.

Locating a reasonable number of participants and co-evolution examples for a comprehensive user study was not feasible in the context of this thesis. Nevertheless, the comparison presented in Section 6.1.5 suggests that productivity might be increased when using dedicated structures for user-driven co-evolution. By demonstrating an approach to user-driven co-evolution that uses dedicated structures, this thesis provides a foundation for further, more rigorous evaluation. For example, the HUTN specification [OMG 2004] makes claims about the human-usability of the notation, but the usability of HUTN has not been studied or compared with other modelling notations. The implementations of the co-evolution structures described in Chapter 5 enable further comparisons.

6.1.7 Summary

This section has demonstrated two approaches to user-driven co-evolution using a co-evolution example from a project in which a graphical model editor was created for process-oriented programs. The first approach used the structures available in EMF alone, while the second approach used two of the structures described in Chapter 5. Comparing the two approaches highlighted differences between the way in which conformance problems were reported and between the modelling notations used to reconcile conformance problems. The comparison described in Section 6.1.5 suggests that developer productivity might be increased by using the second approach, but, as discussed in Section 6.1.6, further work is required to more rigorously evaluate this claim.

6.2 Evaluating Conservative Copy

In contrast to the previous section, this section focuses on developer-driven migration, in which migration is specified in a programming language. As discussed in Chapter 4, the programming languages typically used to specify migration vary and, in particular, use different approaches to relating source and target model elements. This section evaluates the novel source-target relationship implemented in Flock (Section 5.4), *conservative copy*, by comparison to *new-target* and *existing-target* source-target relationships, which have been used for model migration in [Cicchetti *et al.* 2008, Garcés *et al.* 2009]) and [Herrmannsdoerfer *et al.* 2009b, Hussey & Paternostro 2006]) respectively.

The evaluation performed in this section aims to demonstrate that migration strategies are more concise when written with a migration language that uses conservative copy rather than when written with a new- or existing-target migration language. Arguably, more concise migration strategies lead to increased developer productivity (because less code is written to specify migration), and, moreover, to increased understandability of migration strategies (because less code must be read to comprehend a migration strategy).

Conciseness might be measured in many ways. For instance, [Kolovos 2009] counts lines of code to argue that more concise software components indicate a high degree of inter-component re-use. In that context, the number of lines of code is an appropriate measure because the software components were written in a single programming language. [Halstead 1977] suggests ways in which the conciseness and understandability of programs might be approximated by determining the ratio of operators (language constructs) to operands (data). Halstead's Metrics are calculated from programming language constructs and, consequently, are affected by variations in programming languages. Here, counting lines of code and Halstead's metrics are inappropriate because no single language implements the three styles of source-target relationship that are to be compared.

Instead, conciseness was measured by counting the frequency of *model operations*, program statements that are used to manipulate the target (migrated) model. Model operations were specified in a language-independent manner and then mapped onto language-specific constructs to perform the counting. Therefore, the hypothesis for the comparison was: *specifying a migration strategy with conservative copy requires no more model operations than when new-target or when existing-target are used instead.* The results presented in Section 6.2.4 corroborate the hypothesis and highlight some limitations of the implementation of conservative copy in Flock.

The remainder of this section briefly recaps the theoretical differences between the three styles of source-target relationship (Section 6.2.1), describes the co-evolution examples and languages used in the comparison (Section 6.2.2), and details the comparison method (Section 6.2.3). Finally, the results of the comparison (Section 6.2.4) are used to support the claims made above, and to highlight limitations of the conservative copy implementation provided by Flock.

6.2.1 Styles of Source-Target Relationship

Two styles of source-target relationship, new-target and existing-target, are used in existing approaches to model migration, and a third is proposed in this thesis, conservative copy. The differences between the source-target relationships were discussed in Chapter 5 and are now summarised.

With a *new-target* source-target relationship, the migrated model is created afresh by the model migration strategy. The model migration language does not automatically copy any part of the original model to the migrated model. Consequently, any model elements that are not affected by metamodel evolution must be explicitly copied from original to migrated model.

By contrast, the migrated model is initialised to be a copy of the original model in an *existing-target* source-target relationship. Prior to execution of the migration strategy, the migrated and original models are identical. Elements that no longer conform to the evolved metamodel might have been copied automatically from original to migrated model and, consequently, the migration strategy may need to delete model elements.

This thesis proposes a third style of source-target relationship termed *conservative copy*, which is a hybrid of new- and existing-target source-target relationships. Prior to the execution of the migration strategy, only those model elements that conform to the evolved metamodel are copied from original to migrated model. Conservative copy aims to reduce the amount of copying operations often required to specify migration with new-target, while also reducing the amount of delete operations used with existing-target.

6.2.2 Equipment

[1] Five examples of co-evolution taken from three projects, and three reference implementations of source-target relationships were used to perform the comparison described in this section. The co-evolution examples and the selection process for the reference implementations are now discussed.

Co-evolution Examples

To reduce contamination of the comparison, the co-evolution examples used were distinct from those identified in Chapter 4 and subsequently used in the design of Flock in Chapter 5. The examples used for evaluating conservative copy are now summarised, and more details can be found in Appendix C.

Five co-evolution examples taken from three projects were used for evaluating conservative copy. Two examples were taken from the *Newsgroup* project, which performs statistical analysis of NNTP newsgroups, developed by Dimitris Kolovos, a lecturer in this department. One example was taken from changes made to *UML* (the Unified Modeling Language) between versions 1.4 [OMG 2001] and 2.2 [OMG 2007b] of the specification. Two examples were taken from *GMF* (Graphical Modeling Framework) [Gronback 2009], an Eclipse project for generating graphical model editors.

For the newsgroup and GMF projects, the co-evolution examples were identified from source code management systems. The revision history for

[1] TODO: Need a more appropriate name for this section

each project was examined, and metamodel changes were located. The intended migration strategy was determined by speaking with the developer (for the Newsgroup project) and by examining examples and documentation (for GMF). For the UML project, the co-evolution example was identified from the list of changes in the UML 2.2 specification [OMG 2007b], and by discussion with other UML users as described in Section 6.4.

For interoperability with the three reference implementations used in the comparison, the UML co-evolution was adapted. The original (UML 1.4 [OMG 2001]) metamodel is specified in XMI 1.2 [OMG 2007c], which is not supported by two of the reference implementations. The part of the UML 1.4 relating to activity graphs was reconstructed by the author in XMI 2.1 and used in place of the XMI 1.2 version. The reconstructed metamodel was checked by several UML users and was used in the expert evaluation described in Section 6.4, where the reconstructed metamodel is discussed further.

Reference Implementations Used in the Comparison

A formal semantics has not been specified for new-target, existing-target and conservative copy, and therefore the comparison reported in this section was performed using a reference implementation of each source-target relationship. Reference implementations for new- and existing-target were selected from the implementations used by existing approaches to model migration and compared to the implementation of conservative copy provided by Flock.

New-target The Atlas Transformation Language (ATL) is a model-to-model transformation language that has been used in [Cicchetti *et al.* 2008, Garcés *et al.* 2009] for model migration. ATL can be used to specify model migration with new-target, but not with existing-target as discussed in Section 5.3.2. For the comparison described in this section, ATL was selected as the new-target language because the author is not aware of any further approaches to model migration that use an alternative implementation of new-target.

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

COPE was preferred to Ecore2Ecore because the latter is not as expressive[2] and cannot be used for migration in the co-evolution examples considered here.

In summary, the comparison described in this section uses ATL for investigating new-target, Groovy-for-COPE for existing-target, and Flock for conservative copy.

6.2.3 Method

The comparison method is now described. Following the selection of co-evolution examples and reference implementations, the author wrote a migration strategy for each co-evolution example in each of the reference implementations (ATL, Groovy-for-COPE and Flock). The intended migration strategy was determined from models available in the source code management system of the co-evolution example (Newsgroup and GMF projects), or (for the UML project) by referring to the UML specification and discussing ambiguities with other UML users, as described in Section 6.4.

Next, a set of model operations were identified in a language independent manner and then mapped onto language constructs in ATL, Groovy-for-COPE and Flock. The counting of model operations was then automated by implementing a counting program, which was tested and used to further develop the comparison technique. Finally, the counting program was executed on the evaluation examples and the results investigated (Section 6.2.4).

Because the author is more familiar with Flock than with ATL and Groovy-for-COPE, the comparison method has an obvious drawback: the migration strategies written in the latter two languages might be more concise if they were written by the developers of ATL and Groovy-for-COPE. The evolutionary operators built into COPE provide many examples of migration strategy code written by the developer of COPE and, where possible, this code was re-used.

Language-Independent Model Operations

The way in which model operations were identified and counted is now described. Four types of model operation were considered for inclusion in the evaluation: model element creation and deletion operators, and model value assignment and unassignment operators.

Creation and deletion operators are used to create or delete model elements in the migrated model. Assignment and unassignment operators are used to set or unset data values in the migrated model. Typically, assignment operators are used for copying values from the original to the migrated model.

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

Deletion and unassignment operators are not necessary when specifying model migration with new-target, because the migrated model is created afresh by the model migration strategy. Any deletion or unassignment would involve removing model elements or values created explicitly elsewhere in the migration strategy. By contrast, existing-target and conservative copy will automatically create model elements and assign model values prior to the execution of the model migration strategy and hence unassignment and deletion operators are required.

Creation operators were not included in the comparison because, unlike the other operators, they are difficult to specify with regular expressions (and hence automatically count). Moreover, all of the co-evolution examples considered in the comparison assign values to model elements after they are created. Consequently, at least one assignment operator is used whenever a creation operator would have been used.

Model Operations in ATL, Groovy-for-COPE and Flock

The concrete syntax of the deletion, assignment and unassignment model operations in each language is now introduced. First however, it is important to note that the languages considered provide loop constructs and consequently a single model operation might be executed several times during the execution of a migration strategy. Here, a model operation is counted only once even if it is contained in a loop because the comparison reasons about the conciseness of migration strategies, and not about the way in which model operations are executed.

New-target in ATL For new-target in ATL, the following model operation was counted:

- **Assignment:**

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

The assignment operator is used to copy values from the original to the migrated model. Typically, the `value` on the right-hand side is a literal, the value of a feature in the original model, or derived from a combination of the two. Listing 6.1 shows these typical uses of an assignment operator in ATL: line 4 assigns to a literal value, line 5 to the value of a feature in the original model, and line 6 to a value derived from two features in the original model that are separated with a literal value. In the listings in the remainder of this section, lines on which model operations appear are highlighted.

```
1 rule Person2Employee {
2   from o : Before!Person
3   to m : After!Employee (
4     role <- "Unknown",
```

```
    id  <- o.id,
    name <- o.forename + " " + o.surname
  )
}
```

Listing 6.1: Assignment operators in ATL

As discussed above, deletion and unassignment operators are not used for new-target model migration.

Existing-target in Groovy-for-COPE For existing-target in Groovy-for-COPE, the following model operations were counted:

- **Assignment:**

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

- **Unassignment:**

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

- **Deletion:**

```
delete <element>
```

Unlike ATL, Groovy-for-COPE provides distinct operators for assigning to single- and multi-valued features. The first assignment operator assigns to a single-valued feature, the second adds one value to a multi-valued feature, and the third adds multiple values to a multi-valued feature. The fourth form allows the feature name to be determined at runtime and, hence, facilitates reflective access to the model.

COPE provides two forms of unassignment. The first can be used to unassign any feature. The second form is used to remove one value from a multi-valued feature.

Conservative Copy in Epsilon Flock Epsilon Flock, a transformation language tailored for model migration, was developed in this thesis and discussed in Chapter 5. For Flock, the following model operations were counted:

- **Assignment:**

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

- **Unassignment:**

  `<element>.<feature> := null`

  `<element>.<feature>.remove(<value>)`

- **Deleting:**

  `delete <element>`

Like Groovy-for-COPE, Flock distinguishes between assignment to single- and multi-valued features and, hence, provides three assignment operators. Unlike Groovy-for-COPE, Flock does not provide a form of assignment that allows the name of the assigned feature to be determined at runtime.

Flock does not provide a dedicated language construct for performing unassignment, which is instead achieved by assignment to `null`. One value can be removed from a multi-valued feature with the second form of unassignment.

Development and Testing of Method

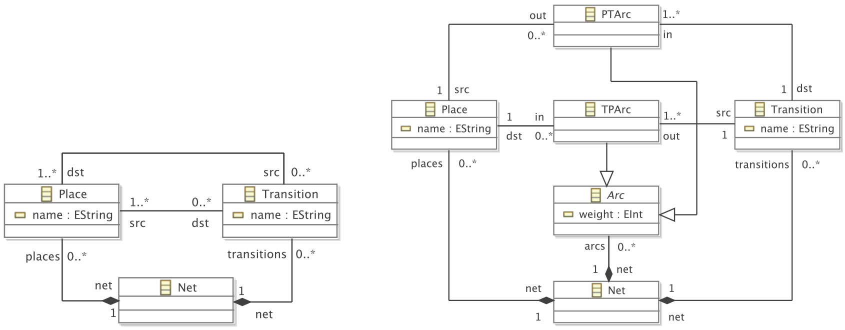
The method and a program for counting model operations were developed and tested by using the co-evolution examples described in Chapter 4, which were used to derive the thesis requirements. An example of model operation counting is given in the remainder of this section, along with the total number of model operations observed for each of the co-evolution examples described in Chapter 4.

Consider the example of metamodel-evolution shown in Figure 6.11. This is the Petri nets metamodel evolution described in Sections 5.3 and 5.4. The migration strategy replaces `Arcs` with `PTArcs` or `TPArcs`. In ATL, the migration strategy uses 12 model operations (Listing 6.2). In Groovy-for-COPE, the migration strategy uses 10 model operations (Listing 6.3) . In Flock, the migration strategy uses 6 model operations (Listing 6.4). These results are also shown in the *(Literature) PetriNets* row of Table 6.1.

Table 6.1 shows the total number of model operations needed to specify migration in ATL, Groovy-for-COPE and Flock for each of the co-evolution examples from Chapter 4. Because the examples used to produce the measurements shown in Table 6.1 were used to design Flock, they are not used to evaluate conservative copy. Instead, they are presented here to show the way in which the evaluation method was developed, and because one of the results (*Refactor: Change Ref to Cont*) highlighted a limitation of the existing-target and conservative copy implementations in COPE and Flock, which is discussed in Section 6.2.4.

6.2.4 Results

By counting the model operations in model migration strategies, the similarities and differences between the three styles of source-target relationship were



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

Figure 6.11: Exemplar metamodel evolution. Taken from [Rose *et al.* 2010e].

	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
(Process-Oriented) Split CP	8	1	1
(Refactor) Cont to Ref	4	5	3
(Refactor) Ref to Cont	3	5	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 6.1: Model operation frequency (analysis examples).

```
rule Nets {
 from o : Before!Net
 to m : After!Net (
   places <- o.places,
   transitions <- o.transitions
 )
}

rule Places {
 from o : Before!Place
 to m : After!Place (
   name <- o.name
 )
}

rule Transitions {
 from o : Before!Transition
 to m : After!Transition (
     name <- o.name,
     "in" <- o.src->collect(p | thisModule.PTArcs(p,o)),
     out <- o.dst->collect(p | thisModule.TPArcs(o,p))
   )
}

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 6.2: The 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 6.3: The Petri nets model migration in Groovy-for-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();
  }
}
```

Listing 6.4: Petri nets model migration in Flock

	Migration Language Source-Target Relationship		
(Project) Example	**ATL** New	**G-f-C** Existing	**Flock** Conservative
(Newsgroup) Extract Person	9	4	3
(Newsgroup) Resolve Replies	8	3	2
(UML) Activity Diagrams	15	15	8
(GMF) Graph	101	10	13
(GMF) Gen2009	310	16	16

Table 6.2: Model operation frequency (evaluation examples).

investigated. The five co-evolution examples discussed in Section 6.2.2 were measured to obtain the results shown in Table 6.2.

The comparison hypothesis stated that *specifying a migration strategy with conservative copy requires no more model operations than when new-target or when existing-target are used instead.* For four of the five examples in Table 6.2, the results support the hypothesis, but the results for the GMF Graph example do not.

The comparison hypothesis did not consider differences between new-target and existing-target, but the results show that, for the most part, a migration strategy uses fewer model operations when using existing-target rather than new-target. For all of the examples in Table 6.2 and most of the examples in Table 6.1, no migration strategy specified with existing-target contained fewer model operations when specified with new-target. However, three of the Refactor examples in Table 6.1 required more model operations when specified with existing-target than when specified with new-target.

The results are now investigated, starting by discussing the way in which the results support the comparison hypothesis. Subsequently, results that contradict the hypothesis are analysed in more detail. The analysis led to the discovery of two limitations of the conservative copy implementation in Flock, which are also discussed below.

Investigation of results

As discussed in Section 6.2.1, 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.

For four of the co-evolution examples, the results in Table 6.2 support the comparison hypothesis. Additionally, the results in Table 6.2 indicate that a

```
rule Canvas2Canvas {
 from o : Before!Canvas
 to m : After!Canvas (
   figures <- o.figures,
   nodes <- o.nodes,
   connections <- o.connections,
   compartments <- o.compartments,
   labels <- o.labels
 )
}
```

Listing 6.5: An extract of the GMF Graph model migration in ATL

migration strategy can be specified with fewer model operations when using existing-target rather than new-target. Three of the results for the Refactor examples in Table 6.1 contradict this contention.

In two of the co-evolution examples, a large proportion of metamodel features were not affected by evolution. These were the GMF examples shown in Table 6.2, which involved evolution of a small part of the metamodels. For the GMF examples, the ATL (new-target) migration strategies use many more model operations than Groovy-for-COPE (existing-target) and Flock (conservative copy). It is likely that the same phenomenon would have been observed, had the actual UML 1.4 metamodel been used for the UML co-evolution example, but this was not possible for the reasons given in Section 6.2.2.

New-target initialises an empty model and, hence, every element of the migrated model must be derived from the original model. For model elements that do not need to be changed in response to metamodel evolution, the migration strategy must copy those elements without change. For instance, the new-target version of the GMF Graph and Gen migration strategies contain many transformation rules such as the one shown in Listing 6.5, which exist only for copying model elements from the original to the migrated model. In Listing 6.5, 5 model operations are used (all assignments) to copy values from the original to the migrated model. The features shown in Listing 6.5 (`figures`, `nodes`, `connections`, `compartments` and `labels`) were not changed during metamodel evolution. Unlike new-target, existing-target and conservative copy do not require explicit copying of model elements from the original to migrated model due to the way in which they initialise the migrated model.

In the UML co-evolution example (Table 6.2) and the Refactor Inline Class (Table 6.1), a large proportion of metamodel features were renamed. For these examples, expressing migration with an existing-target transformation language requires more model operations than using a new-target transformation language.

Existing-target requires two model operations be used when a feature is renamed, while new-target and conservative copy require only one model operation. For instance, the `transitions` feature of `ActivityGraph` was renamed to `edge` in the UML co-evolution example. The code used for migration in response to this change for new-target, existing-target and conservative copy is shown below.

New-target: `edge <- transitions`
Existing-target: `element.edge = element.unset(transitions)`
Conservative copy: `migrated.edge := original.transitions`

As shown above, migration in response to feature renaming typically requires one model operation when using new-target and conservative copy (an assignment). When using existing-target, the equivalent migration strategy requires an additional model operation (an unassignment) that removes the value from the old feature. Note that, in Groovy-for-COPE, the `unset` function unassigns a feature and returns the (unassigned) value.

The results in Table 6.2 corroborate the comparison hypothesis for four of the five examples. The five migration strategies required no more model operations when specified with conservative copy than when specified with new- and existing-target. When specified with conservative copy, the migration strategies did not contain explicit copying (which was required when using new-target for the GMF examples) and used one rather than two model operations for migration in response to feature renaming (which required two model operations when using existing-target). The GMF Graph co-evolution example does not support the hypothesis due to a limitation of the way in which conservative copy is implemented in Flock. This limitation is described below.

Two conclusions can be drawn from this discussion. Firstly, in general, fewer model operations are used when specifying a migration strategy with a conservative copy migration language than when specifying the same migration strategy with a new- or existing-target migration language. Secondly, in the examples studied here, there are often more features unaffected by metamodel evolution than affected. Consequently, specifying model migration with a new-target migration language requires more model operations than in an existing-target migration language for the examples shown in Tables 6.1 and 6.2. [Sprinkle 2003] suggests that metamodel evolution often involves changes to relatively few metamodel elements, and the results presented in this section support his contention.

Limitation 1: Duplication when migrating subtypes

For the GMF Graph example (Table 6.2), conservative copy requires more model operations than existing-target. Investigation of this result revealed a limitation of the conservative copy implementation provided by Flock, which is now described and illustrated using a simplification of the GMF Graph co-evolution example.

Figure 6.12 shows part of the GMF Graph metamodel prior to evolution. In the real metamodel, the `figure` and `accessor` features are references to other metamodel classes, rather than attributes. When the metamodel evolved, the types of the `figure` and `accessor` features were changed. Here, let us assume that their types were changed from a string to an integer. The actual metamodel changes are described in Section C.3.1.

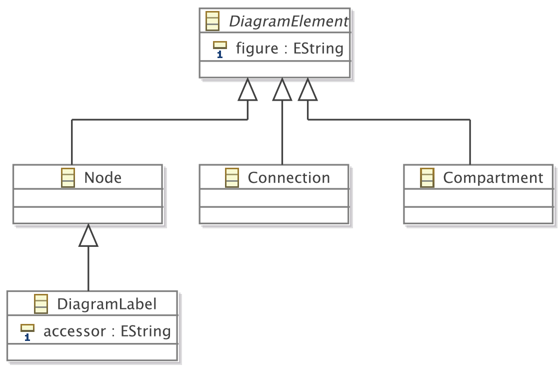


Figure 6.12: Simplified fragment of the GMF Graph metamodel.

In response to re-typing of the `figure` and `accessor` features, the migration strategy derived new values for the `figure` and `accessor` features. In the real example, a new model element was created and used to decorate [Gamma *et al.* 1995] each old value. In the simplified example presented here, the new integer value will be derived from the old string value by using its length. Section C.3.1 presents the strategies used to perform migration for the actual metamodel changes.

As demonstrated below, ATL and Groovy-for-COPE provide mechanisms for re-using migration code between subtypes. Migration of the `figure` feature can be specified once and used for migrating all subtypes of `DiagramElement`. Currently, Flock does not provide a mechanism for re-using migration code between subtypes.

In ATL (Listing 6.6), the GMF Graph migration strategy was expressed using two model operations: the two assignment operations on lines 3 and 23. For `Nodes`, `Connections` and `Compartments`, migration of the `figure` feature is achieved by extending the `DiagramElement` transformation rule. Note the use of the `extends` keyword on lines 6, 11 and 16 for inheriting the rule on lines 1-4. For `DiagramLabels`, the values of both the `accessor` and `figure` features must be migrated. On lines 21-24, the `DiagramLabels` extends `Nodes` and hence `DiagramElements` to inherit the body of the

```
abstract rule DiagramElements {
  from o : Before!DiagramElement
  to  m : After!DiagramElement (
    figure <- o.figure.length()
  )
}

rule Nodes extends DiagramElements {
  from o : Before!Node
  to  m : After!Node
}

rule Connections extends DiagramElements {
  from o : Before!Connection
  to  m : After!Connection
}

rule Compartments extends DiagramElements {
  from o : Before!Compartment
  to  m : After!Compartment
}

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

Listing 6.6: Simplified GMF Graph model migration in ATL

latter for migrating figures, and, in addition, the `DiagramLabels` rule defines the migration for the value of the `accessor` feature.

In Groovy-for-COPE (Listing 6.7), the migration is similar to ATL but is specified imperatively. In Listing 6.7, a loop iterates over each instance of `DiagramElement` (line 1), migrating the value of its figure feature (line 2). The `allInstances` function is used to locate every model element with the type `DiagramElement` or one of its subtypes. If the `DiagramElement` is also a `DiagramLabel` (line 4), the value of its accessor feature is also migrated (line 5). In Groovy-for-COPE, the migration strategy uses two model operations: the assignment statements on lines 2 and 5.

In both ATL and Groovy-for-COPE, only 2 model operations are required for this migration: an assignment for each of the two features being migrated. However, the equivalent Flock migration strategy, shown in Listing 6.8, requires 5 model operations: the assignment statements on lines 2,

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

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

Listing 6.7: Simplified GMF Graph model migration in COPE

```
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 6.8: Simplified GMF Graph model migration in Flock

6, 10, 11 and 15. Note that the migration of the `figure` feature is specified four times (once for each subtype of `DiagramElement`). A single `DiagramElement` rule cannot be used to migrate the `figure` feature because, when a `migrate` rule does not specify a `to` part, Flock will create an instance of the type named after the `migrate` keyword. In other words, a `migrate DiagramElement` rule will result in Flock attempting to instantiate the abstract class `DiagramElement`. Instead migration must be specified using four migrate rules, as shown in Listing 6.8.

In the current implementation of Flock, `migrate` rules are used for specifying two concerns and the limitation described here might be avoided if those concerns were specified using two distinct language constructs. Firstly, the `to` part of a `migrate` rule is used to establish type equivalences between the original and evolved metamodel. When a metaclass is renamed, for example, migration in Flock would typically use a rule of the form `migrate`

OldType to NewType. Omitting the to part of a rule (migrate X) is a shorthand for migrate X to X. Secondly, the body of each rule specifies the way in which each model element should be migrated. Separating the two concerns using distinct language constructs might facilitate the re-use of migration code between subtypes, as is the case in ATL and Groovy-for-COPE. The extent to which greater re-use and increased conciseness can be addressed with changes to the implementation of Flock, and in general, is discussed in Section 7.2. This section now considers one further limitation of existing-target and conservative copy migration languages.

Limitation 2: Side-effects during initialisation

The measurements observed for one of the examples of co-evolution from Chapter 4, Change Reference to Containment (Table 6.1), 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 6.13). A Signature references any number of ports. The metamodel is evolved to prevent the sharing of Ports between Signatures by changing the ports feature to a containment rather than a reference (Figure 6.14). Ports are contained in Signatures rather than in Systems, and consequently the ports is no longer a feature of System.

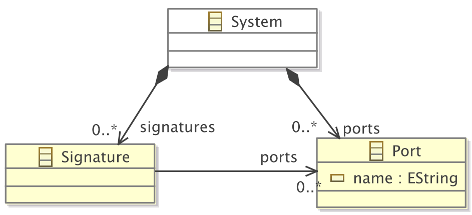


Figure 6.13: Original metamodel.

Listing 6.9 shows the migration strategy using new-target in ATL. Three model operations are used: the assignment statements on lines 3, 8 and 14. The rules for migrating Systems (lines 1-4) and Ports (13-15) copy values for the features unaffected by evolution (signatures and name respectively). The rule for migrating Signatures (lines 6-11) clones each member of the ports feature (using the Port rule on lines 13-15). Crucially, the Ports

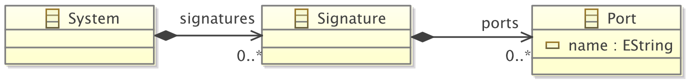


Figure 6.14: Evolved metamodel.

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

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

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

Listing 6.9: Migration for Change Reference to Containment in ATL

rule is marked as `lazy` and consequently is only executed when called from the `Signatures` rule. By contrast, the `Systems` and `Signatures` rules are executed automatically by ATL for each `System` and `Signature` in the original model, respectively.

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

In COPE, the containment nature of a reference is not enforced until after

```
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 6.10: Migration for Change Reference to Containment in Groovy-for-COPE

the migration strategy is executed. Hence, migration for the type of metamodel change considered here can be specified by unsetting the contents of the `ports` reference (line 4 of Listing 6.10), and creating a copy of each referenced `Port` (lines 5-7 of Listing 6.10). The migration strategy shown in Listing 6.10 uses 5 model operations: assignments on lines 8, 9 and 12; unassignment on line 10, and deletion on line 19. Note that the assignment on line 7 is to a temporary variable, and so is not counted as a model operation.

The ATL migration strategy copies ports as they are needed. By contrast, the Groovy-for-COPE migration strategy must account for the initial copy of every Port which is created prior to execution according to existing-target. The Groovy-for-COPE migration strategy must either clone only those ports that are referenced by more than one signature, or clone every referenced port, but delete all of the ports created by existing-target. The latter approach requires 1 more model operation (to delete the original ports) than the former (shown in Listing 6.10).

In Flock, the containment nature of the reference is enforced when the migrated model is initialised. Because changing the contents of a containment reference has side-effects, a `Port` that appears in the `ports` reference of a

```
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:
12   not Original!Signature.all.exists(s|s.ports.includes(original))
```

Listing 6.11: Migration for Change Reference to Containment in Flock

`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 6.11). The Flock migration strategy uses 3 model operations: assignments on lines 5 and 6, and a deletion on 11.

The Groovy-for-COPE and Flock migration strategies must remove any `Ports` which are not referenced by any `Signature` (lines 17-21 of Listing 6.10, and line 11 of Listing 6.11 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, migration strategies written in Flock and Groovy-for-COPE must account for the side-effects that can occur during initialisation of the migrated model, resulting in less concise migration strategies. The existing-target and conservative copy implementations used in COPE and Flock might be changed to avoid this limitation by either automatically cloning values when a reference is changed to be a containment reference, or by allowing the user to specify features that should not be copied by the source-target relationship during initialisation. Section 7.2 discusses this issue further.

6.2.5 Summary

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

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

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

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

6.3 Evaluating Co-evolution Tools

This section compares four co-evolution tools to determine their strengths and weaknesses and to provide guidance for selecting a model migration tool. As discussed in Chapter 3, several tools for managing co-evolution are described in the literature. Chapter 5 introduces a further tool, Flock. While each tool has strengths and weaknesses, little is known about how migration tools compare in practice.

The comparison and guidance presented in this section aim to simplify tool selection by recommending tools for particular situations or requirements. For example, when scalability is a concern (many large models are to be migrated), the advice presented in this section indicates that a developer should consider AML and Ecore2Ecore. Furthermore, the comparison highlighted situations in which Flock is more or less suitable than other tools, as discussed in Section 6.3.3.

The comparison reported in this section was performed using an expert evaluation. Flock and three further co-evolution tools, selected from those described in Chapter 4, were compared by MDE experts. Following the process outlined in Section 6.3.1, the tools were applied to two co-evolution examples. The remainder of this section reports the experts' experiences with each tool (Section 6.3.2), and synthesises advice and guidelines for identifying the most appropriate model migration tool in different situations (Section 6.3.3).

This section is based on joint work with Markus Herrmannsdöerfer (a research student at Technische Universität München), James Williams (a research student in this department), Dimitrios Kolovos (a lecturer in

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

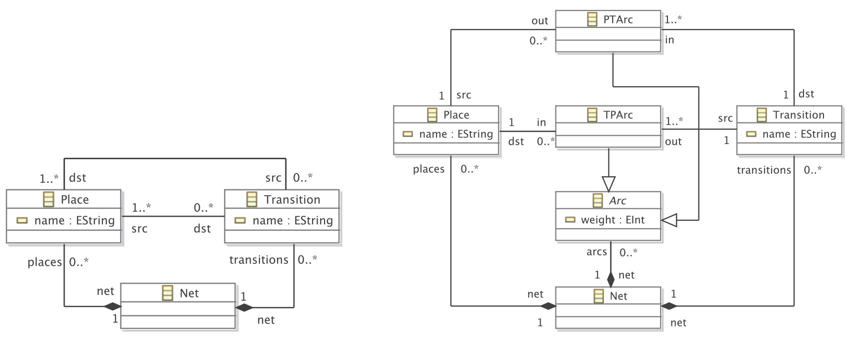
6.3.1 Comparison Method

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

Co-Evolution Examples

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

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



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

Figure 6.15: Petri nets metamodel evolution (taken from [Rose *et al.* 2010e]).

In Figure 6.15(a), a Petri `Net` comprises `Places` and `Transitions`. A `Place` has any number of `src` or `dst` `Transitions`. Similarly, a `Transition` has at least one `src` and `dst` `Place`. In this example, the metamodel in Figure 6.15(a) is evolved to support weighted connections between `Places` and `Transitions` and between `Transitions` and `Places`.

The evolved metamodel is shown in Figure 6.15(b). `Places` are connected to `Transitions` via instances of `PTArc`. Likewise, `Transitions` are connected to `Places` via `TPArc`. Both `PTArc` and `TPArc` inherit from `Arc`, and therefore can be used to specify a `weight`.

GMF. The second example is taken from GMF [Gronback 2009], an Eclipse project for generating graphical editors for models. The development of GMF is model-driven and utilises four domain-specific metamodels. Here, we consider one of those metamodels, GMF Graph, and its evolution between GMF versions 1.0 and 2.0. The GMF Graph example is now summarised, and more details can be found in Section C.3.1.

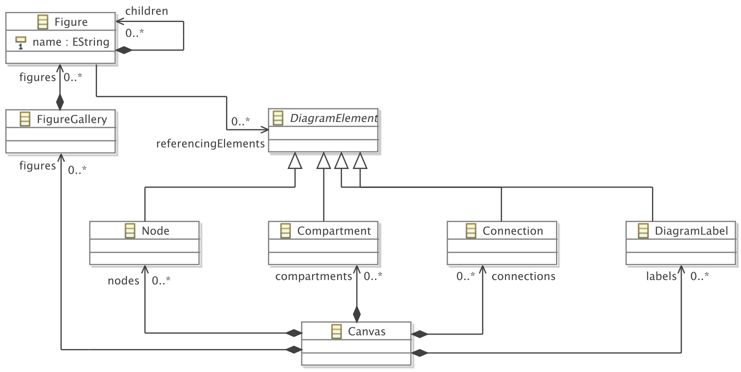
The GMF Graph metamodel (Figure 6.16) describes the appearance of the generated graphical model editor. As described in the GMF Graph documentation[3], the Graph metamodel from GMF 1.0 was evolved – as shown in Figure 6.16(b) – to facilitate greater re-use of figures by introducing a proxy [Gamma *et al.* 1995] for `Figure`, termed `FigureDescriptor`. The original `referencingElements` reference was removed, and an extra metaclass, `ChildAccess` in its place. Section C.3.1 discusses the metamodel changes in more detail.

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

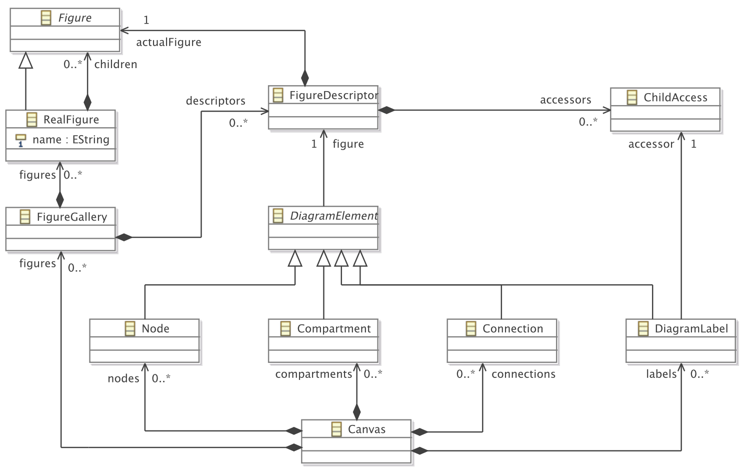
Compared Tools

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

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



(a) Original metamodel.



(b) Evolved metamodel.

Figure 6.16: GMF graph metamodel evolution

widely used modelling framework. AML, COPE and Ecore2Ecore were discussed in Chapter 4, and Epsilon Flock in Chapter 5.

Comparison Process

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

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

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

Table 6.3: Summary of comparison criteria.

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

6.3.2 Comparison Results

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

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

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

Constructing the migration strategy

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

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

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

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

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

Changing the migration strategy

Migration strategies can change in at least two ways. Firstly, as a migration strategy is developed, testing might reveal errors which need to be corrected. Secondly, further metamodel changes might require changes to an existing migration strategy.

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

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

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

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

Extensibility

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

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

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

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.

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

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

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

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

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

Conciseness

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

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

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

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

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

Clarity

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

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

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

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

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

Expressiveness

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

AML. A complete migrating transformation could be generated for the Petri nets example, but not for the GMF Graph example. The latter contains examples of two complex changes that AML does not currently support[6]. Success-

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

fully expressing the GMF Graph example in AML would require changes to at least one of AML's heuristics. However, AML provided an initial migration transformation that was completed by hand.

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

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

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

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

Interoperability

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

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

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

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

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

Performance

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

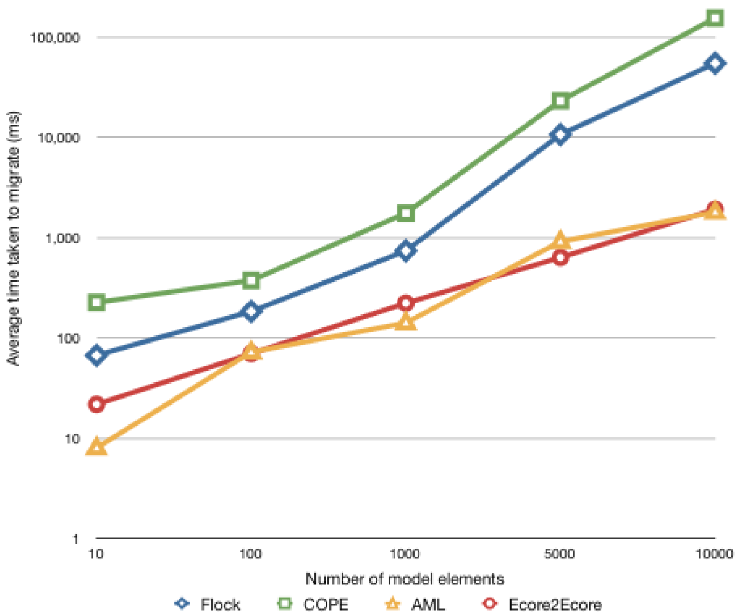


Figure 6.17: Migration tool performance comparison.

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

To measure performance, five sets of Petri net models were generated at random. Models in each set contained 10, 100, 1000, 5,000, and 10,000 model elements. Figure 6.17 shows the average time taken by each tool to execute migration across 10 repetitions for models of different sizes. Note that the Y axis has a logarithmic scale. The results indicate that, for the Petri nets co-evolution example, AML and Ecore2Ecore execute migration significantly more quickly than COPE and Flock, particularly when the model to be migrated contains more than 1,000 model elements. Figure 6.17 indicates that, for the Petri nets co-evolution example, Flock executes migration between two and three times faster than COPE, although the author of COPE reports that turning off validation causes COPE to perform similarly to Flock.

6.3.3 Discussion

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

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

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

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

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

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

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

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

Requirement	**Recommended Tools**
Frequent, incremental co-evolution	COPE
Reverse-engineering	AML, Ecore2Ecore, Flock
Modelling technology diversity	Flock
Quicker migration for larger models	AML, Ecore2 Ecore
Minimal dependencies	Ecore2Ecore
Minimal hand-written code	AML, COPE
Minimal guidance from user	AML
Support for metamodel-specific migrations	COPE, Flock

Strengths and Weaknesses of Flock

The comparison and guidance highlight strengths and weaknesses of AML, COPE, Ecore2Ecore and Flock. The findings for Flock are now summarised.

Strengths Flock was the only co-evolution tool suitable for performing model migration when the original and evolved metamodels are specified in different modelling technologies. For the examples of metamodel evolution explored here, Flock is more expressive than AML, but requires more guidance from the user. This is consistent with the trade-off between flexibility and level of automation of co-evolution approaches identified in Chapter 4.

Weaknesses The results presented here indicate that model migration with Flock takes longer to execute than with AML and Ecore2Ecore. This is likely because Flock migration strategies are interpreted, while AML and Ecore2Ecore migration strategies are compiled. Compared to COPE and AML, Flock lacks re-use of model migration patterns across varying metamodels. In Flock, model migration is specified in terms of concrete metamodel types and cannot be re-used for different metamodels. By contrast, COPE and AML capture model migration in a metamodel-independent manner.

6.3.4 Summary

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

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

6.4 Transformation Tools Contest

In contrast to the previous section, which compared Flock to three co-evolution tools, the evaluation performed in this section compares Flock with model-to-model transformation tools. As discussed in Chapter 4, model migration can be regarded as a specialisation of model-to-model transformation. Chapter 5 introduces Flock, a language tailored for model migration. This section assess the suitability of Flock for specifying model migration and for specifying model-to-model transformation by comparison to other model-to-model transformation languages.

To this end, the author participated in the 2010 edition of the Transformation Tools Contest (TTC), a workshop series that seeks to compare and contrast tools for performing model and graph transformation. At TTC 2010[8], two rounds of submissions were invited: cases (transformation problems, three of which are selected by the workshop organisers) and solutions to the selected

[8] `http://www.planet-research20.org/ttc2010/index.php?Itemid=132`

cases. Nine transformation tools, including Flock, were assessed for a model migration problem based on a real-world example of metamodel evolution from the UML [OMG 2007b].

Compared to the evaluation described in Section 6.3, the evaluation in this section compares Flock to a wider range of tools (model and graph transformation tools, and not just model migration tools), and investigates the suitability of Flock for specifying model transformation (and not just model migration). The remainder of this section describes the model migration problem (Section 6.4.1) and Flock solution (Section 6.4.2).

6.4.1 Model Migration Case

To compare Flock with other transformation tools for specifying model migration, the author submitted a case to TTC based on the evolution of the UML. The way in which activity diagrams are modelled in the UML changed significantly between versions 1.4 and 2.1 of the specification. In the former, activities were defined as a special case of state machines, while in the latter they are defined atop a more general semantic base[9] [Selic 2005].

The remainder of this section briefly introduces UML activity diagrams, describes their evolution, and discusses the way in which solutions were assessed. Section C.2.1 describes the metamodel evolution in more detail. The work presented in this section is based on the case submitted to TTC 2010 [Rose *et al.* 2010d].

Activity Diagrams in UML

Activity diagrams are used for modelling lower-level behaviours, emphasising sequencing and co-ordination conditions. They are used to model business processes and logic [OMG 2007b]. Figure 6.18 shows an activity diagram for filling orders. The diagrams is partitioned into three *swimlanes*, representing different organisational units. *Activities* are represented with rounded rectangles and *transitions* with directed arrows. *Fork* and *join* nodes are specified using a solid black rectangle. *Decision* nodes are represented with a diamond. Guards on transitions are specified using square brackets. For example, in Figure 6.18 the transition to the restock activity is guarded by the condition `[not in stock]`. Text on transitions that is not enclosed in square brackets represents a trigger event. In Figure 6.18, the transition from the restock activity occurs on receipt of the asynchronous signal called `receive stock`. Finally, the transitions between activities might involve interaction with objects. In Figure 6.18, the Fill Order activity leads to an interaction with an object called `Filled Object`.

[9] A variant of generalised coloured Petri nets.

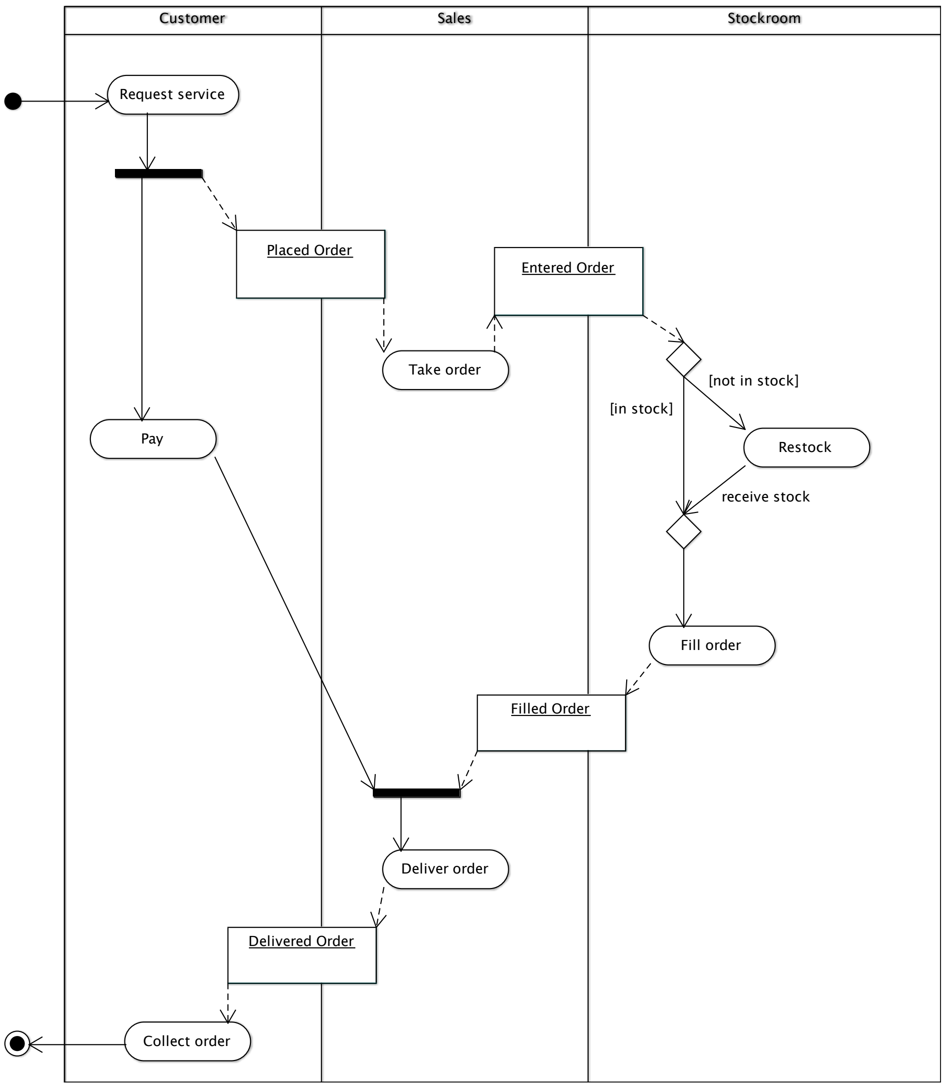


Figure 6.18: Exemplar activity model.

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

Evolution of Activity Diagrams

Figures 6.19 and 6.20 are simplifications of the activity diagram metamodels from versions 1.4 and 2.2 of the UML specification, respectively. In the interest of clarity, some features and abstract classes have been removed from Figures 6.19 and 6.20.

Some differences between Figures 6.19 and 6.20 are: activities have been changed such that they comprise nodes and edges, actions replace states in UML 2.2, and the subtypes of control node replace pseudostates.

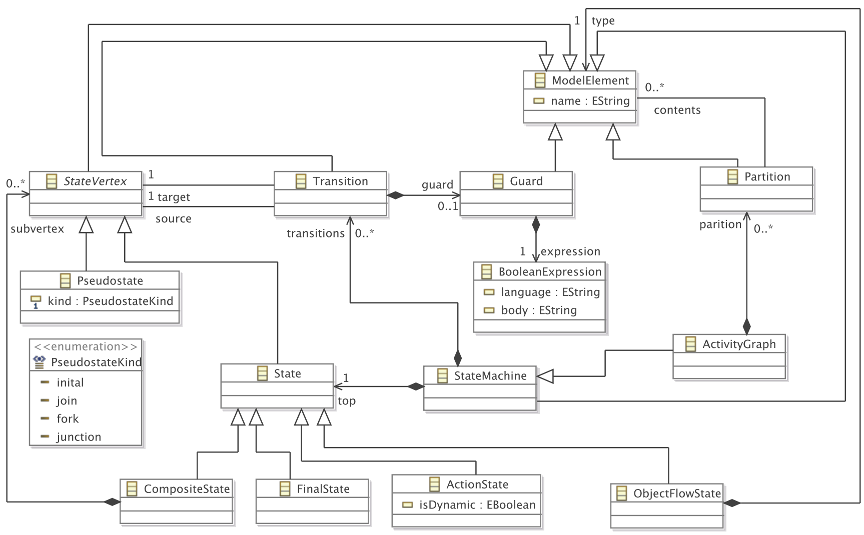


Figure 6.19: UML 1.4 Activity Graphs (based on [OMG 2001]).

To facilitate the comparison of solutions, the model shown in Figure 6.18 was used. Figure 6.18 is based on [OMG 2001, pg3-165]. Solutions migrated the activity diagram shown in Figure 6.18 – which conforms to UML 1.4 – to conform to UML 2.2. The UML 1.4 model, the migrated UML 2.2 model, and the UML 1.4 and 2.2 metamodels are available from[10].

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

- **Correctness**: Does the transformation produce a model equivalent to the migrated UML 2.2. model included in the case resources?

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

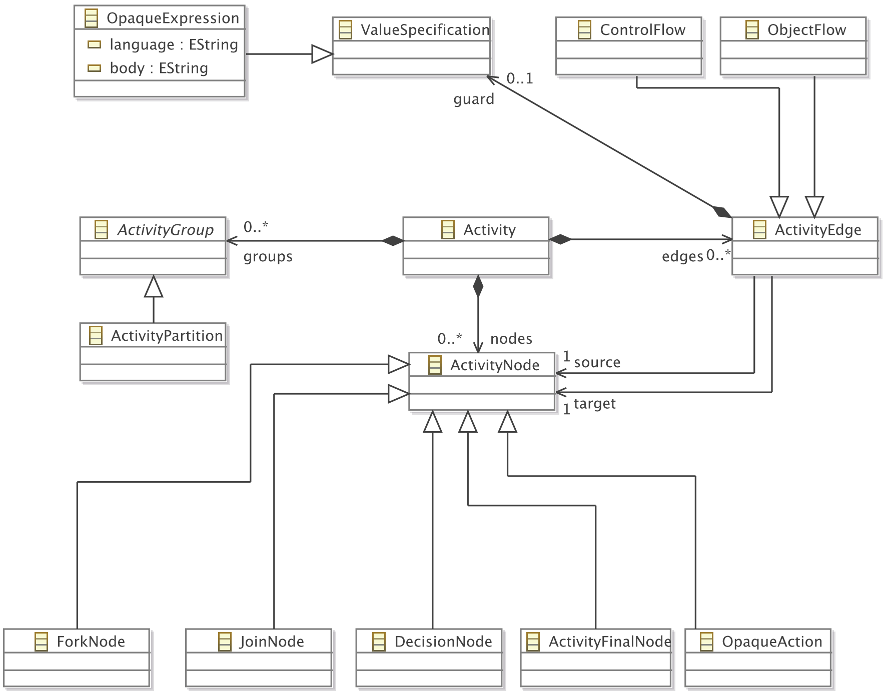


Figure 6.20: UML 2.2 Activity Diagrams (based on [OMG 2007b]).

- **Conciseness**: How much code is required to specify the transformation? (In [Sprinkle & Karsai 2004] et al. propose that the amount of effort required to codify migration should be directly proportional to the number of changes between original and evolved metamodel).

- **Clarity**: How easy is it to read and understand the used transformation? (For example, is a well-known or standardised language?)

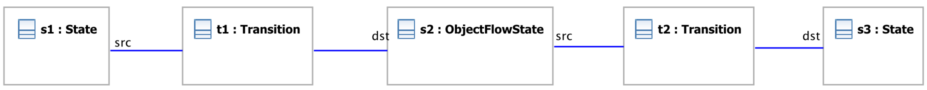
- **Extensions**: Which of the case extensions (described below) were implemented in the solution?

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

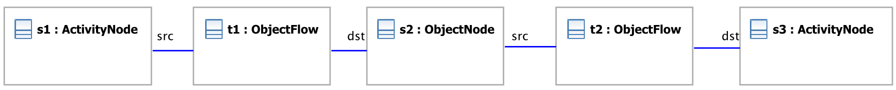
Extension 1: Alternative Object Flow State Migration Semantics

Following the submission of the case, discussion on the TTC forums[11] revealed an ambiguity in the UML 2.2 specification indicating that the migration semantics for the ObjectFlowState UML 1.4 concept are not clear from the UML 2.2 specification. The case was revised to incorporate both the original semantics (suggested by the author and described above) and an alternative semantics (suggested by a workshop participant via the TTC forums) for migrating ObjectFlowStates. The alternative semantics are now described.

In the core task described above, instances of `ObjectFlowState` were migrated to instances of `ObjectNode`. Any instances of `Transition` that had an `ObjectFlowState` as their source or target were migrated to instances of `ObjectFlow`. Figure 6.21 shows an example application of this migration semantics. Structures such as the one shown in Figure 6.21(a) are migrated to an equivalent structure shown in Figure 6.21(b). The `Transitions`, `t1` and `t2`, are migrated to instances of `ObjectFlow`. Likewise, the instance of `ObjectFlowState`, `s2`, is migrated to an instance of `ObjectNode`.



(a) ObjectFlowState structure in UML 1.4



(b) Equivalent ObjectNode structure in UML 2.2

Figure 6.21: Migrating Actions for the Core Task

This extension considered an alternative migration semantics for ObjectFlowState. For this extension, instances of `ObjectFlowState` (and any connected `Transitions`) were migrated to instances of `ObjectFlow`, as shown in Figure 6.22 in which the UML 2.2 `ObjectFlow`, `f1`, replaces `t1`, `t2` and `s2`.

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

(a) ObjectFlowState structure in UML 1.4

(b) Equivalent ObjectFlow structure in UML 2.2

Figure 6.22: Migrating Actions for Extension 1

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

Extension 2: Concrete Syntax

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

Submissions were invited to explore the feasibility of migrating the concrete syntax of the activity diagram shown in Figure 6.18 to the concrete syntax in their chosen UML 2 tool. To facilitate this, the case resources included an ArgoUML project[12] containing the activity diagram shown in Figure 6.18.

Extension 3: XMI

The UML specifications [OMG 2001, OMG 2007b] indicate that UML models should be stored using XMI. However, because XMI has evolved at the same time as UML, UML 1.4 tools most likely produce XMI of a different version to UML 2.2 tools. For instance, ArgoUML produces XMI 1.2 for UML 1.4 models, while the Eclipse UML2 tools produce XMI 2.1 for UML 2.2.

As an extension to the core task, submissions were invited to consider how to migrate a UML 1.4 model represented in XMI 1.x to a UML 2.1 model

[12] `http://argouml.tigris.org/`

represented in XMI 2.x. To facilitate this, the UML 1.4 model shown in Figure 6.18 was made available in XMI 1.2 as part of the case resources.

Following the submission of the case, Tom Morris, the project leader for ArgoEclipse and a committer on ArgoUML, encouraged solutions to consider the extension described above. ArgoUML cannot, at present, migrate models from UML 1 to UML 2. On the TTC forums, Morris stated that "We have nothing available to fill this hole currently, so any contributions would be hugely valuable. Not only would achieve academic fame and glory from the contest, but you'd get to see your code benefit users of one of the oldest (10+ yrs) open source UML modeling tools." [13]

6.4.2 Model Migration Solution in Epsilon Flock

This section describes a Flock solution for migrating UML activity diagrams in response to the evolution described above. The solution was developed by the author, and, at the workshop, compared with migration strategies written in other languages. The workshop participants and organisers rated each tool.

The Flock migration strategy was developed in an iterative and incremental manner, using the following process, starting with an empty migration strategy:

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

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

Actions, Transitions and Final States

Development of the migration strategy began by executing an empty Flock migration strategy on the original model. Because Flock automatically copies model elements that have not been affected by evolution, the resulting model contained `Pseudostatess` and `Transitions`, but none of the `ActionStates` from the original model. In UML 2.2 activities, `OpaqueActions` replace `ActionStates`. Listing 6.12 shows the Flock code for changing `ActionStates` to corresponding `OpaqueActions`.

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

```
migrate ActionState to OpaqueAction
```

Listing 6.12: Migrating Actions

Next, similar rules were added to migrate instances of `FinalState` to instances of `ActivityFinalNode` and to migrate instances of `Transition` to `ControlFlow`, as shown in Listing 6.13.

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

Listing 6.13: Migrating FinalStates and Transitions

Pseudostates

Development continued by selected further types of state that were not present in the migrated model, such as `Pseudostatess`, which are not used in UML 2.2 activities. Instead, UML 2.2 activities use specialised `Nodes`, such as `InitialNode`. Listing 6.14 shows the Flock code used to change `Pseudostates` to corresponding `Nodes`.

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

Listing 6.14: Migrating Pseudostates

Activities

In UML 2.2, `Activitys` no longer inherit from state machines. As such, some of the features defined by `Activity` have been renamed. Specifically, `transitions` has become `edges` and `paritions` has become `group`. Furthermore, the states (or nodes in UML 2.2 parlance) of an `Activity` are now contained in a feature called `nodes`, rather than in the `subvertex` feature of a composite state accessed via the `top` feature of `Activity`. The Flock migration rule shown in Listing 6.15 captured these changes.

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

Listing 6.15: Migrating ActivityGraphs

Note that the rule in Listing 6.15 used the built-in `equivalent` operation to find migrated model elements from original model elements. As discussed in Section 5.4, the `equivalent` operation invokes other migration rules where necessary and caches results to improve performance.

Next, a similar rule for migrating `Guards` was added. In UML 1.4, the the `guard` feature of `Transition` references a `Guard`, which in turn references an `Expression` via its `expression` feature. In UML 2.2, the `guard` feature of `Transition` references an `OpaqueExpression` directly. Listing 6.16 captures this in Flock.

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

Listing 6.16: Migrating Guards

Partitions

In UML 1.4 activity diagrams, `Partition` specifies a single containment reference for its `contents`. In UML 2.2 activity diagrams, partitions have been renamed to `ActivityPartitions` and specify two containment features for their contents, `edges` and `nodes`. Listing 6.17 shows the rule used to migrate `Partitions` to `ActivityPartitions` in Flock. The body of the rule shown in Listing 6.17 uses the *collect* operation to segregate the `contents` feature of the original model element into two parts.

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

Listing 6.17: Migrating Partitions

ObjectFlows

Finally, two rules were written for migrating model elements relating to object flows. In UML 1.4 activity diagrams, object flows are specified using `ObjectFlowState`, a subtype of `StateVertex`. In UML 2.2 activity diagrams, object flows are modelled using a subtype of `ObjectNode`. In UML 2.2 flows that connect to and from `ObjectNodes` must be represented with `ObjectFlows` rather than `ControlFlows`.

Listing 6.18 shows the Flock rule used to migrate `Transitons` to `ObjectFlows`. The rule applies for `Transitions` whose source or target `StateVertex` is of type `ObjectFlowState`.

```
migrate ObjectFlowState to ActivityParameterNode

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

Listing 6.18: Migrating ObjectFlows

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

Alternative ObjectFlowState Migration Semantics

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

Listing 6.19 shows the Flock source code used to migrate `ObjectFlowStates` (and connecting `Transitions`) to a single `ObjectFlow`. This rule was used instead of the two rules defined in Listing 6.18. In the body of the rule shown in Listing 6.19, the `source` of the `Transition` is copied directly to the `source` of the `ObjectFlow`. The `target` of the `ObjectFlow` is set to the target of the first outgoing `Transition` from the `ObjectFlowState`.

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

Listing 6.19: Migrating ObjectFlowStates to a single ObjectFlow

Because, in this alternative semantics, `ObjectFlowStates` are represented as edges rather than nodes, the partition migration rule was changed such that `ObjectFlowStates` were not copied to the `nodes` feature of `Partitions`. To filter out the `ObjectFlowStates`, line 3 of Listing 6.17 was changed to include a reject statement, as shown on line 3 of Listing 6.20.

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

Listing 6.20: Migrating Partitions without ObjectFlowStates

The complete source code listing for the Flock migration strategy is provided in Section C.2.1.

XMI

The second extension required submissions to migrate an activity graph conforming to UML 1.4 and encoded in XMI 1.2 to an equivalent activity graph conforming to UML 2.2 and encoded in XMI 2.1. The core task did not require submissions to consider changes to XMI (the model storage representation), but, in practice, this is a challenge to migration, as noted by Tom Morris on the TTC forums[14].

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

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

Comparison with other solutions

At the workshop, solutions to the migration case described in Section 6.4.1 were presented. Each solution was allocated two opponents who highlighted weaknesses of each approach. Following the solution presentations and opposition statements, each solution was scored using the four criteria described above, correctness, clarity, conciseness and number of extensions solved. Epsilon Flock was awarded first position for the migration case. The opposition statements for Flock and the solution scores are now discussed.

Opposition Statements The opposition statements highlighted two weaknesses of Flock. Firstly, there is some duplicated code in Listing 6.14: the `migrate Pseudostate to ...` statement appears several times. The duplication exists because Flock only allows one-to-one mappings between original and evolved metamodel types. The conservative copy algorithm would need to be extended to allow one-to-many mappings to remove this kind of duplication.

Secondly, the body of Flock rules are specified in an imperative manner. Consequently, reasoning about the correctness of the a migration strategy is arguably more difficult than in languages that use a purely declarative syntax.

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

This point is discussed further in Section 6.5, which considers the limitations of the thesis.

Scoring Every workshop participants scored each solution on clarity and conciseness. The workshop organisers scored each solution on correctness and number of extensions solved, as these criteria could be measured objectively. Flock was awarded the most points by the workshop participants and organisers. The complete list of scores is shown in Table TODO[15].

TODO: Discuss the results: to what extent and in what regard is Flock "better" than the other solutions? Appraise the ranking system.

6.4.3 Summary

This section has discussed the way in which Flock was evaluated by participating in the 2010 edition of the Transformation Tools Contest (TTC). Flock was assessed by application to an example of migration from the UML and comparison with eight other model and graph transformation tools. Flock was awarded first prize by the workshop participants and organisers.

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

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

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

6.5 Limitations

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

[15]TODO: Fill in when Pieter mails the spreadsheet

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

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

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

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

6.6 Summary

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

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

Appendix B

A Graphical Editor for Process-Oriented Programs

This appendix describes the way in which a prototypical graphical editor for process-oriented programs was designed and implemented. User-driven co-evolution, the process identified in Chapter 4, was used to manage evolutionary change. The work presented here was conducted in collaboration with Adam Sampson, then a Research Associate at the University of Kent. The way in which the graphical editor changed throughout its development provided was used for evaluation of the thesis research in Section 6.1.

The purpose of the collaboration was to explore the suitability of MDE tools and techniques for designing a graphical notation and supporting tools for programs written in process-oriented programming languages, such as occam-π [Welch & Barnes 2005]. The collaboration resulted in the design and implementation of a prototypical graphical editor, which was used to construct examples of small process-oriented programs.

Process-oriented programs are specified in terms of three core concepts: processes, connection points and channels. Processes are the fundamental building blocks of a process-oriented program. Channels are the mechanism by which processes communicate, and are unidirectional. Connection points define the channels on which a process can communicate. Connection points are used to specify the way in which a process can communicate, and can optionally be bound to a channel. Because channels are unidirectional, connection points are either reading (consume messages from the channel) or writing (generate messages on the channel).

MDE tools and techniques were used to design and implement a graphical notation and editor for process-oriented programs by Sampson and the thesis author. An iterative style of development was used. The abstract syntax of the domain was specified as a metamodel, captured in Ecore, which is the metamodelling language of arguably the most widely-used contemporary MDE development environment, the Eclipse Modeling Framework (EMF)

[Steinberg *et al.* 2008]. The graphical concrete syntax was specified with the Graphical Modeling Framework (GMF) [Gronback 2006], via EuGENia, [Kolovos *et al.* 2009]. EMF and GMF are described more thoroughly in Section 2.3.

The remainder of this appendix presents the six versions of the process-oriented metamodel constructed by Sampson and the thesis author during the iterative development of the graphical editor. In addition, several models are also shown, which were used to test the graphical editor throughout its development.

B.1 Iteration 1: Processes and Channels

Development began by identifying two key concepts for modelling process-oriented programs. From examples of process-oriented programs, process and channel were identified as the most important concepts, and consequently the metamodel shown in Figure B.1 was constructed.

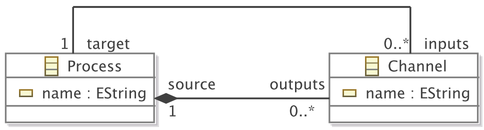


Figure B.1: The process-oriented metamodel after the first iteration.

Additionally, a graphical concrete syntax was chosen for processes and channels. The former were represented as boxes, and the latter as lines. EuGENia annotations were added to the metamodel, resulting in the metamodel shown in Listing B.1. Line 1 of Listing B.1 uses the “@gmf.node” EuGENia annotation to indicate that processes are to be represented as boxes with a label equal to the value of the `name` feature. Line 9 uses the “@gmf.link” EuGENia annotation to indicate that channels are to be represented as lines between `source` and `target` processes with a label equal to the value of the `name` feature.

```
1  @gmf.node(label="name")
2  class Process {
3    attr String name;
4
5    ref Channel[*]#target inputs;
6    val Channel[*]#source outputs;
7  }
8
9  @gmf.link(source="source", target="target", label="name")
10 class Channel {
11   attr String name;
```

```
  ref Process[1]#outputs source;
  ref Process[1]#inputs target;
}
```

Listing B.1: The annotated process-oriented metamodel after the first iteration

To generate code for the graphical editor, EuGENia was invoked on the annotated metamodel shown in Listing B.1. However, EuGENia failed with an error, because no “root” element had been specified. GMF, the graphical modelling framework used by EuGENia, requires one metaclass (termed the root) to be specified as a container for all diagram elements. The root metaclass cannot be a GMF node or a link, and so the second iteration involved adding an additional metaclass for interoperability with GMF.

B.2 Iteration 2: Interoperability with GMF

In the second iteration, an additional metaclass, `Model`, was added to the metamodel as shown in Figure B.2. The `Model` metaclass was used to provide GMF with a container for storing all of the diagram elements for each process-oriented diagram.

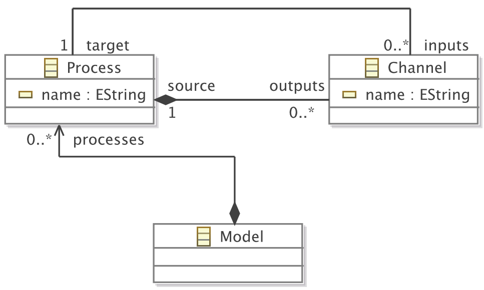


Figure B.2: The process-oriented metamodel after the second iteration.

As shown in Listing B.1, the `Model` metaclass was annotated with “@gmf.diagram” to indicate that it should be used as the diagram’s root element. Root elements do not have a concrete syntax and do not appear in the graphical editor.

```
@gmf.diagram
class Model {
  val Process[*] processes;
}

@gmf.node(label="name")
class Process {
```

```
  attr String name;

  ref Channel[*]#target inputs;
  val Channel[*]#source outputs;
}


@gmf.link(source="source", target="target", label="name")
class Channel {
  attr String name;
  ref Process[1]#outputs source;
  ref Process[1]#inputs target;
}
```

Listing B.2: The annotated process-oriented metamodel after the second iteration

EuGENia was invoked on the annotated metamodel shown in Listing B.2 to produce code for the graphical editor. Figure B.3 shows a simplistic test model, which was constructed to test the generated editor. The model shown in Figure B.3 comprises two processes, `P1` and `P2`, and one channel, `a`.

B.3 Iteration 3: Shared Channels

In previous iterations, channels had been contained within their source process. The nested structure made it more difficult to explore process-oriented models in EMF's tree editor due to the additional level of nesting. Consequently, the metamodel was changed such that channels were contained in the root element, rather than in the source process, resulting in the metamodel shown in Figure B.4.

No additional EuGENia annotations were added to the metamodel during this iteration. In other words, the graphical notation (concrete syntax) was not changed, and the resulting editor was identical in appearance to the previous one. However, the EMF tree editor showed just one level of nesting (everything is contained inside model).

The existing models required migration because of the way in which XMI differentiates between reference and containment values. Each channel was instantiated in the new `channels` reference of `Model`, and existing values in the `outputs` reference of `ConnectionPoint` were changed to use a reference rather than a containment value. Figure B.5(a) shows the HUTN for a model prior to migration, and Listing B.5(b) shows the reconciled, migrated HUTN.

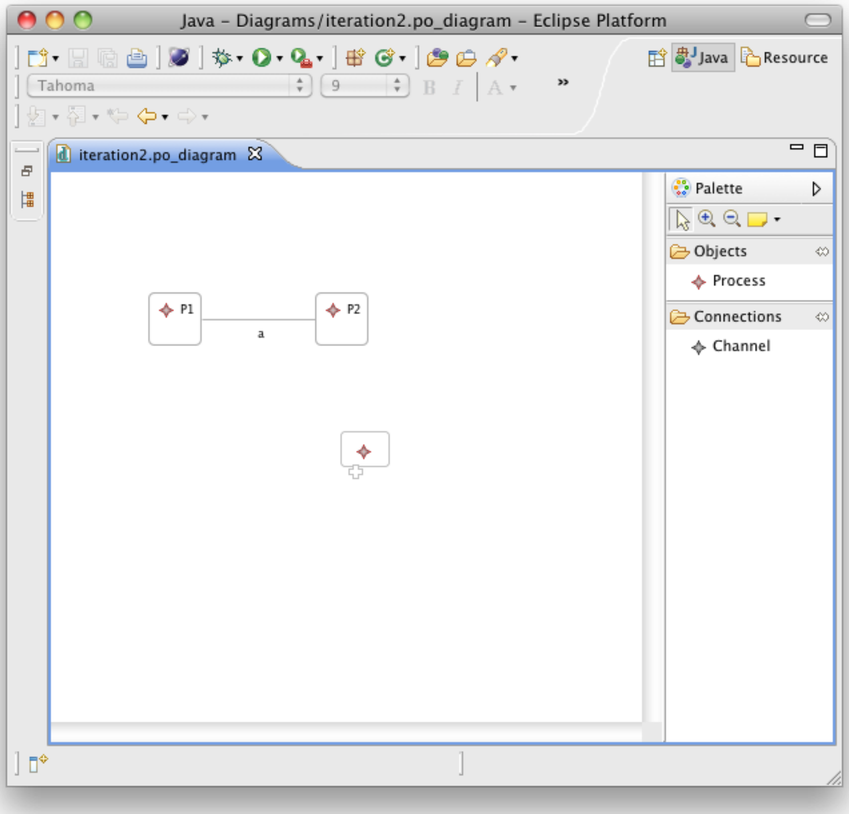


Figure B.3: Exemplar diagram after the second iteration.

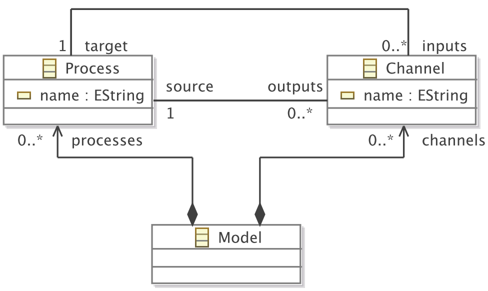
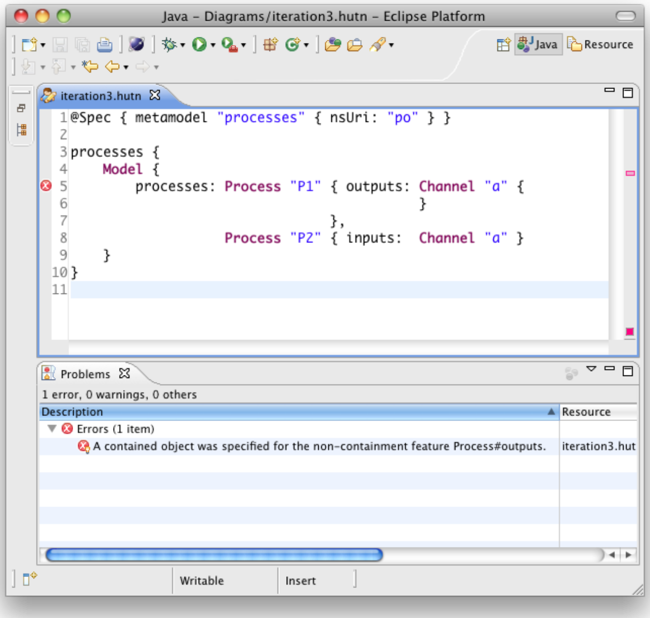
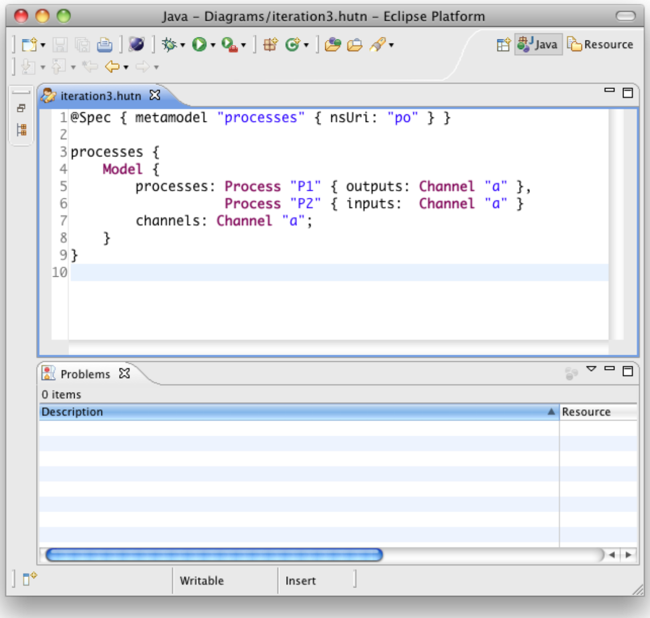


Figure B.4: The process-oriented metamodel after the third iteration.



(a) HUTN prior to migration



(b) HUTN after migration

Figure B.5: Exemplar migration between the second and third versions of the process-oriented metamodel

B.4 Iteration 4: Connection Points

The fourth iteration involved capturing a third domain concept, connection points, in the graphical notation. When a process is specified, the ways in which it can communicate are declared as connection points. When a process is instantiated, channels are connected to its connection points, and messages flow in and out of the process. The graphical notation was to be used to describe both instantiated processes and types of process, the metamodel was changed to model connection points.

The iteration resulted in the metamodel shown in Figure B.6. `ConnectionPoint` was introduced as an association class between the `inputs` and `outputs` references.

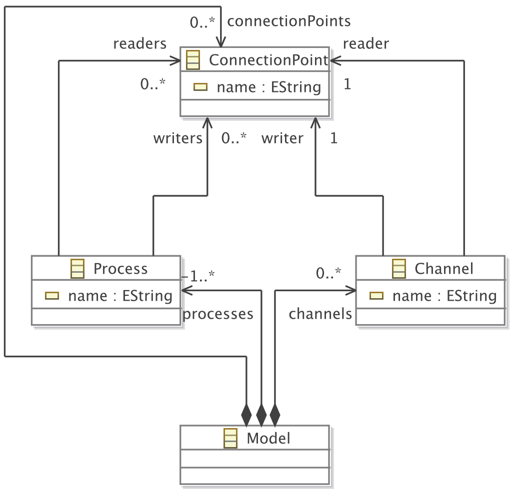


Figure B.6: The process-oriented metamodel after the fourth iteration.

To specify concrete syntax for connection points, additional EuGENia annotations were added to the metamodel as shown in Listing B.3. The `ConnectionPoint` class was annotated with a “@gmf.node” to specify that connections points were to be represented as circles, labelled with the value of the `name` attribute. The circles were to be affixed to the boxes used to represent processes, and, hence, “@gmf.affixed” annotations are used on lines 12 and 15.

```
1  @gmf.diagram
2  class Model {
3    val Process[*] processes;
4    val Channel[*] channels;
5    val ConnectionPoint[*] connectionPoints;
6  }
7
```

```
@gmf.node(label="name")
class Process {
  attr String name;

  @gmf.affixed
  ref ConnectionPoint[*] readers;

  @gmf.affixed
  ref ConnectionPoint[*] writers;
}


@gmf.link(source="reader", target="writer", label="name", incoming="
    true")
class Channel {
 attr String name;
  ref ConnectionPoint[1] reader;
  ref ConnectionPoint[1] writer;
}

@gmf.node(label="name", label.placement="external", label.icon="false",
     figure="ellipse", size="15,15")
class ConnectionPoint {
  attr String name;
}
```

Listing B.3: The annotated process-oriented metamodel after the fourth iteration

A new version of the graphical editor was generated by invoking EuGENia on the annotated metamodel. A larger test model was constructed to test the editor, and is shown in Figure B.7. The existing models required migration because the `inputs` and `outputs` references of `Process` and the `source` and `target` references of `Channel` had been removed.

To migrate each existing model, two connection points were created for each channel in the model. The `source` and `target` reference of the channel was changed to reference the new connection points, as were the corresponding values of the `readers` and `writers` references of the relevant processes. Figure B.8(a) shows the HUTN for a model prior to migration, and Figure B.8(b) shows the reconciled, migrated HUTN.

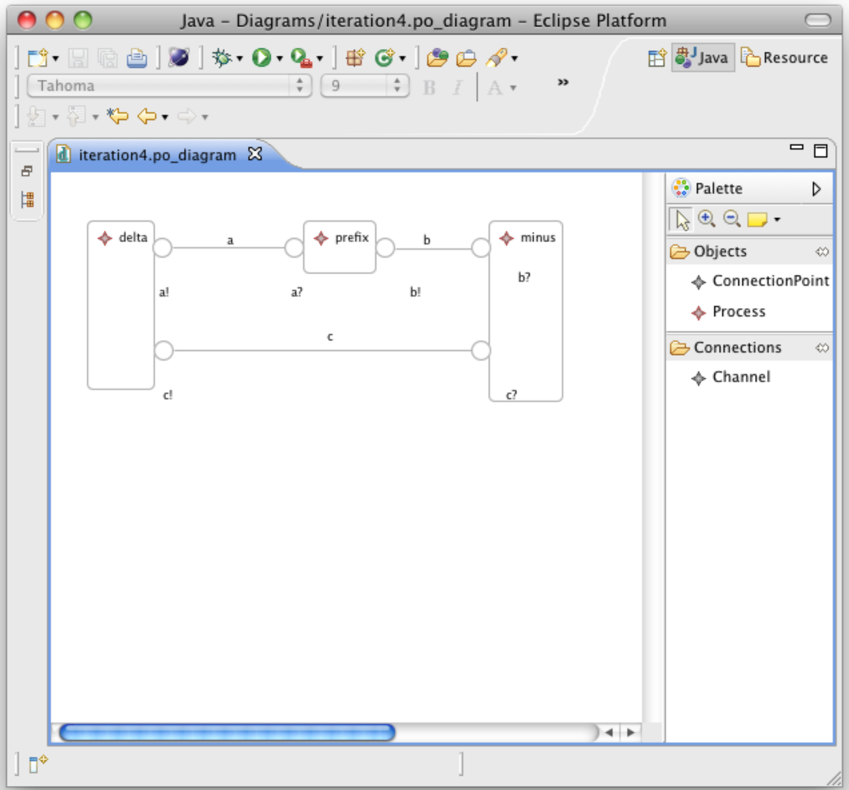
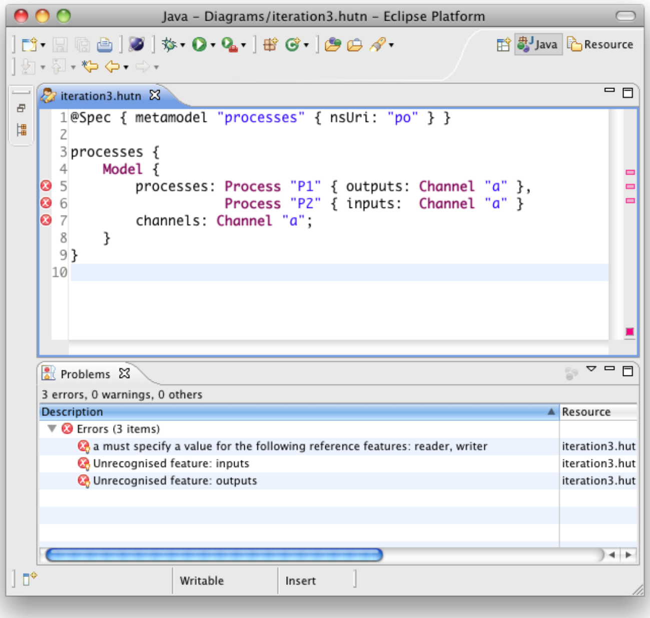
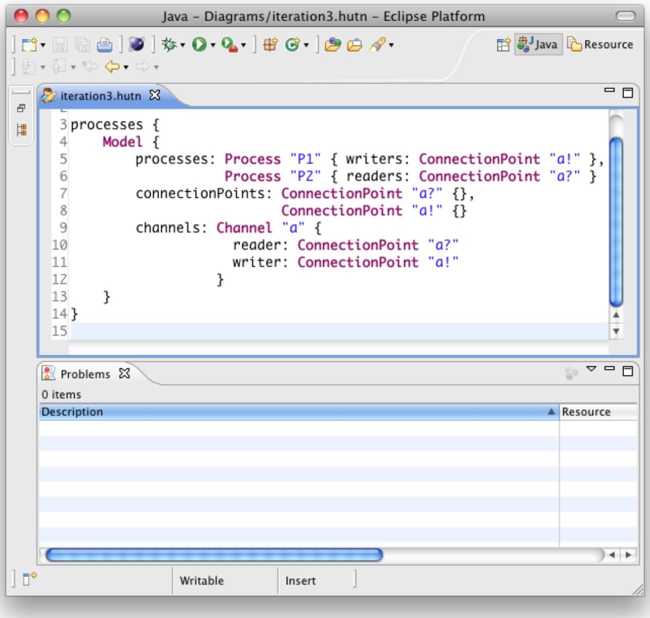


Figure B.7: Exemplar diagram after the fourth iteration.



(a) HUTN prior to migration



(b) HUTN after migration

Figure B.8: Exemplar migration between the third and fourth versions of the process-oriented metamodel

B.5 Iteration 5: Connection Point Types

Channels are unidirectional, and so connection points are either *reading* or *writing*. A process uses the former to consume messages from a channel, and the latter to produce messages on a channel. Testing the graphical editor producing in the fourth iteration showed that it was not immediately obvious as to which connection points were reading and which were writing. The fifth iteration involved changing the graphical editor to better distinguish between reading and writing connection points.

The iteration resulted in the metamodel shown in Figure B.9. `ConnectionPoint` was made abstract, and two subclass, `ReadingConnectionPoint` and `WritingConnectionPoint`, were introduced. The four references to `ConnectionPoint` were changed to reference one of the two subclasses.

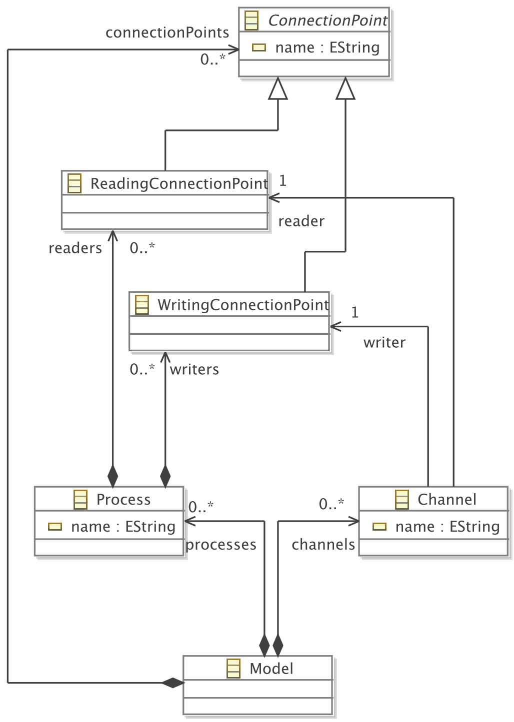


Figure B.9: The process-oriented metamodel after the fifth iteration.

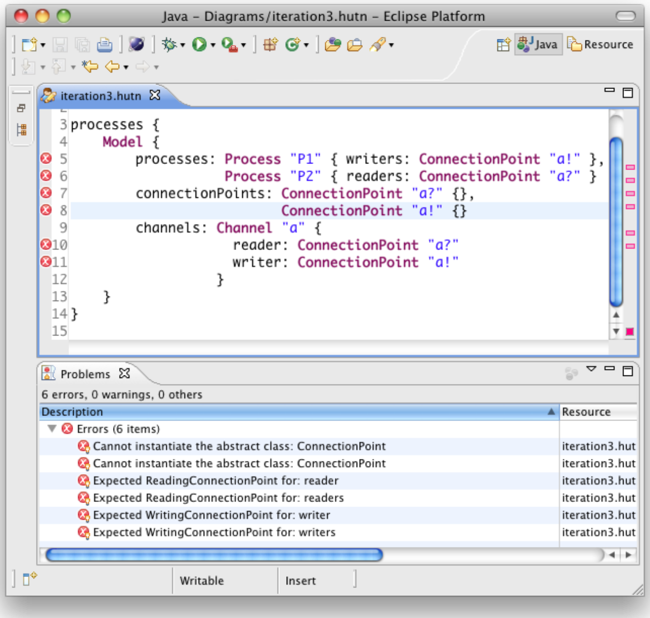
The graphical notation was changed, as shown in Listing B.4. The `WritingConnectionPoint` class was annotated with an additional colour attribute to specify that writing connection points were to be represented with a black circle. White is the default colour for a "@gmf.node" annotation, and so reading connection points were represented as white circles.

```
@gmf.diagram
```

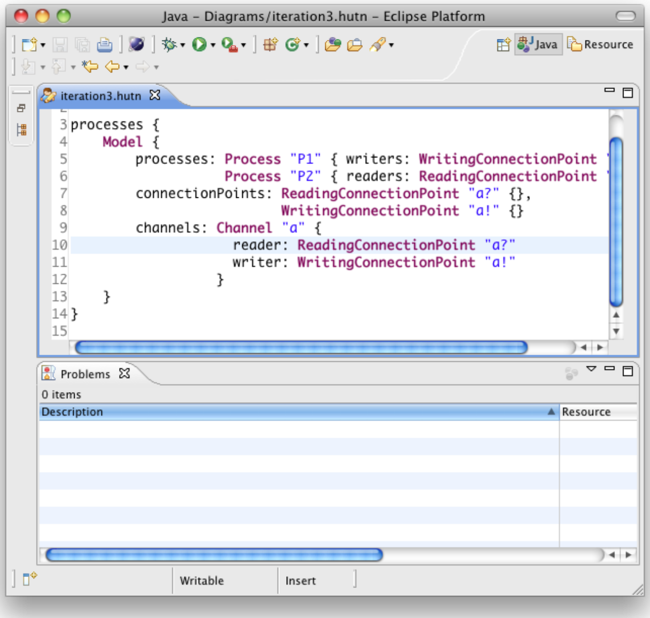
```
class Model {
  val Process[*] processes;
  val Channel[*] channels;
  val ConnectionPoint[*] connectionPoints;
}

@gmf.node(label="name")
class Process {
  attr String name;

  @gmf.affixed
  ref ReadingConnectionPoint[*] readers;

  @gmf.affixed
  ref WritingConnectionPoint[*] writers;
}


@gmf.link(source="reader", target="writer", label="name", incoming="true")
class Channel {
 attr String name;
  ref ReadingConnectionPoint[1] reader;
  ref WritingConnectionPoint[1] writer;
}

@gmf.node(label="name", label.placement="external", label.icon="false", figure="ellipse", size="15,15")
abstract class ConnectionPoint {
  attr String name;
}

class ReadingConnectionPoint extends ConnectionPoint {}

@gmf.node(color="0,0,0")
class WritingConnectionPoint extends ConnectionPoint {}
```

Listing B.4: The annotated process-oriented metamodel after the fifth iteration

A new version of the graphical editor was generated by invoking EuGENia on the annotated metamodel. All of the existing models required migration, because `ConnectionPoint` was now an abstract class, and could no longer be instantiated. Section 6.1 describes the way in which models were migrated after the changes made during this iteration. Briefly, migration involved replacing every instantiation of `ConnectionPoint` with an instantiation of either `ReadingConnectionPoint` or `WritingConnectionPoint`. The former was used when a connection point was used as the value of a channel's `reader` feature and the latter when when a connection point was used as the value of a channel's `writer` feature. Listing B.10(a) shows the HUTN for a model prior to migration, and Listing B.10(b) shows the reconciled, migrated HUTN.



(a) HUTN prior to migration



(b) HUTN after migration

Figure B.10: Exemplar migration between the fourth and fifth versions of the process-oriented metamodel

B.6 Iteration 6: Nested Processes and Channels

The final iteration involved changing the graphical editor such that processes and channels could be nested inside other processes. In some process-oriented languages, such as occam-π [Welch & Barnes 2005], processes can be specified in terms of other, internal processes.

To support the decomposition of processes into other processes and channels, the `nestedProcess` and `nestedChannel` references were added to the `Process` class, as shown in Figure B.11.

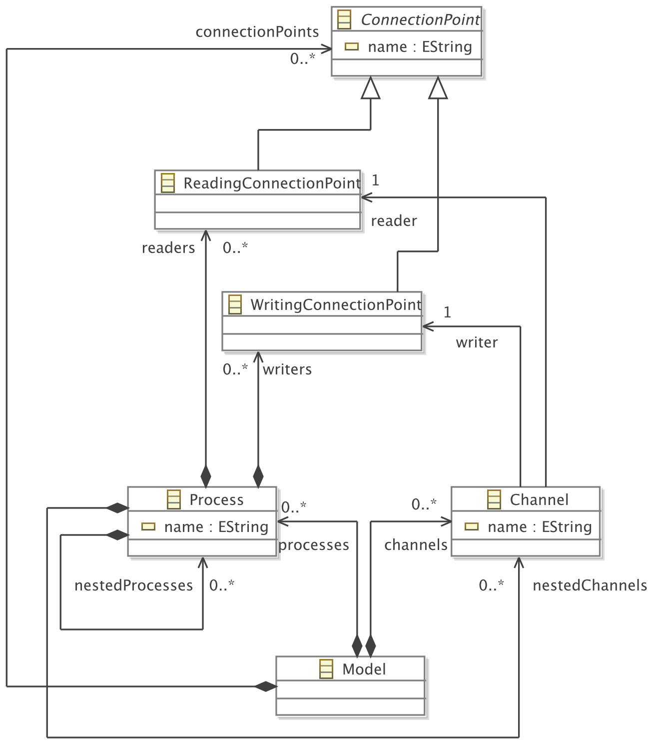


Figure B.11: The process-oriented metamodel after the final iteration.

As shown in Listing B.5, the "@gmf.compartment" annotation was added to the `nestedProcess` to indicate that processes can be placed inside other processes in the graphical editor.

```
@gmf.diagram
class Model {
  val Process[*] processes;
  val Channel[*] channels;
  val ConnectionPoint[*] connectionPoints;
}

```

```
@gmf.node(label="name")
class Process {
  attr String name;

  @gmf.compartment
  val Process[*] nestedProcesses;
  val Channel[*] nestedChannels;

  @gmf.affixed
  ref ReadingConnectionPoint[*] readers;

  @gmf.affixed
  ref WritingConnectionPoint[*] writers;
}


@gmf.link(source="reader", target="writer", label="name", incoming="true")
class Channel {
 attr String name;
  ref ReadingConnectionPoint[1] reader;
  ref WritingConnectionPoint[1] writer;
}

@gmf.node(label="name", label.placement="external", label.icon="false", figure="ellipse", size="15,15")
abstract class ConnectionPoint {
  attr String name;
}

class ReadingConnectionPoint extends ConnectionPoint {}

@gmf.node(color="0,0,0")
class WritingConnectionPoint extends ConnectionPoint {}
```

Listing B.5: The annotated process-oriented metamodel after the final iteration

EuGENia was invoked on the annotated metamodel to produce the final version of the graphical editor. An additional model was constructed to check the nesting of processes, and is shown in Figure B.12. Because the changes made to the metamodel in this iteration involved only adding new features, no migration of existing models was necessary.

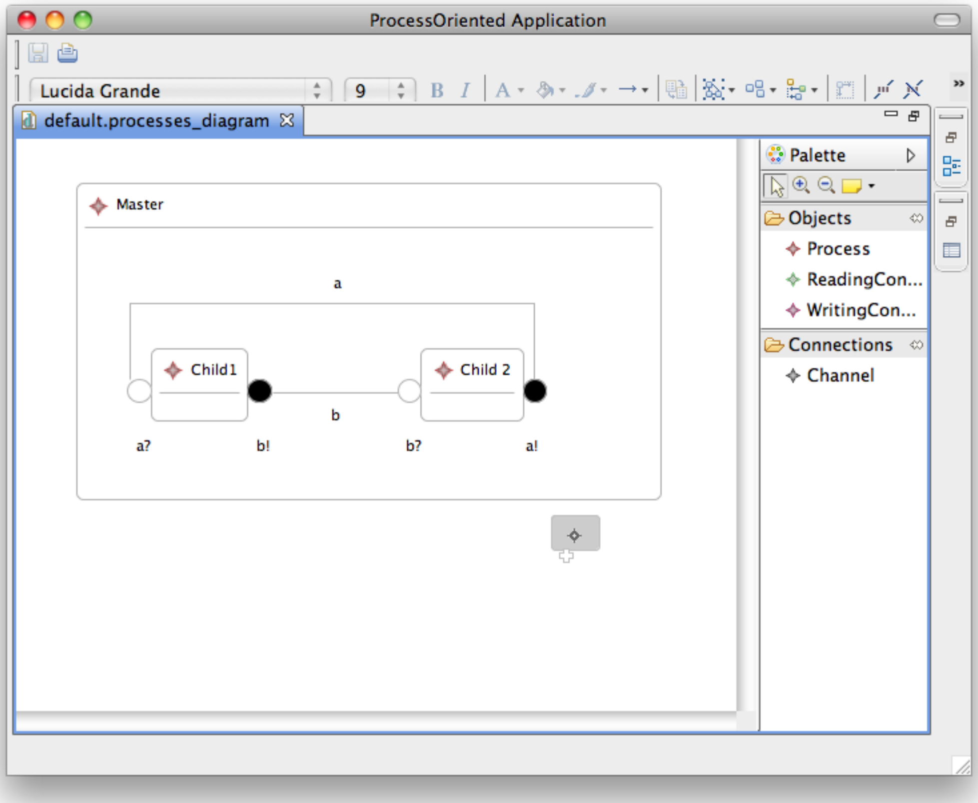


Figure B.12: Exemplar diagram after the final iteration.

B.7 Summary

This appendix has described the way in which a graphical editor for process-oriented programs was designed and implemented using an iterative style of development. A metamodel was used to capture the key concepts of the domain, and to generate code for a graphical editor. Each iteration involved changing the metamodel either to correct unintended behaviour in the editor (iterations 3 and 5), to facilitate interoperability with other tools (iteration 2) or to add new features (iterations 1, 4 and 6). The metamodel changes described in the fifth iteration are used for evaluation of the thesis research in Section 6.1.

Appendix C

Co-evolution Examples

This appendix describes the co-evolution examples used for evaluation in Chapter 6. The examples were taken from real-world MDE projects and are distinct from the examples used for analysis in Chapter 4. Each section details examples from one project, describing metamodel changes and model migration strategies. In accordance with the evaluation performed in Section 6.2, each model migration strategy is presented in three model migration languages. Lines of code that contain *a model operation* (a statement that changes the migrated model) are highlighted. Section 6.2 describes model operation and the three model migration languages in more detail.

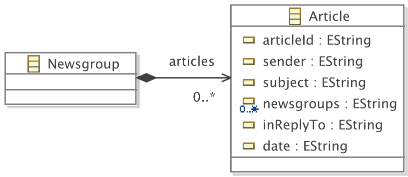
C.1 Newsgroups Examples

This section describes a program for performing statistical analysis of NNTP newsgroups, developed by Dimitris Kolovos, a lecturer in this department. The program comprises a metamodel to capture domain-specific concepts, a text-to-model transformation for parsing newsgroup messages, and a model-to-model transformation for analysing the messages.

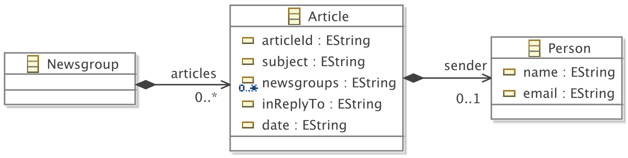
The metamodel and transformations were developed in an iterative and incremental manner. Five iterations of the metamodel and transformations were made available by Kolovos, two of which involved metamodel changes that affected the conformance of existing models. In the other three iterations, the metamodel changes were additive and did not lead to model migration.

C.1.1 Extract Person

The Extract Person iteration involved separating the domain concepts of authors and articles. At the start of the iteration, the `Article` class defined a string attribute called `sender` as shown in Figure C.1(a). To make it easier to recognise articles written by the same people, the `Person` class was



(a) Original metamodel.



(b) Evolved metamodel.

Figure C.1: Newsgroups metamodel during the Extract Person iteration

introduced, and the `sender` attribute was replaced with a reference to the `Person` class as shown in Figure C.1(b).

Existing models were migrated by deriving a `Person` object from the `se-nd-er` feature of each `Article`. The values of the `send-er` feature used one of two forms: `username@domain.com (Full Name)` or `"Full Name" username@domain.com`.

Listings C.1, C.2 and C.3 show the model migration strategy in ATL, COPE and Flock respectively. The `toEmail()` and `toName()` operations are used to extract names and email addresses, are defined without using any model operations, and are omitted from the listings below.

```
module ExtractPerson;

create Migrated : After from Original : Before;

rule Newsgroups {
  from
    o : Before!Newsgroup
  to
    m : After!Newsgroup (
      articles <- o.articles
    )
}

rule Articles {
  from
    o : Before!Article
  to
    m : After!Article (
```

```
    articleId <- o.articleId,
    subject  <- o.subject,
    newsgroups <- o.newsgroups,
    inReplyTo <- o.inReplyTo,
    date    <- o.date,
    sender  <- p
  ),
  p : After!Person (
    name <- o.sender.toName(),
    email <- o.sender.toEmail()
  )
}
```

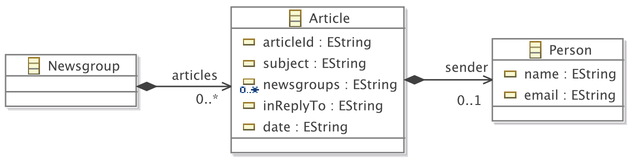
Listing C.1: The Newsgroup Extract Person model migration in ATL

```
toPerson = { str ->
 def person = personClass.newInstance();

 person.email = str.toEmail()
 person.name = str.toName()

 return person
}

for (article in extractperson.Article.allInstances) {
 def sender = article.unset(sender)
 article.sender = toPerson(sender)
}
```

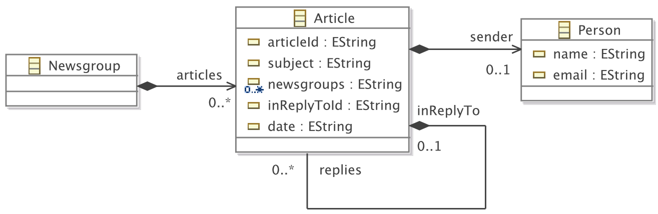
Listing C.2: The Newsgroup Extract Person model migration in COPE

```
migrate Article {
 migrated.sender := original.sender.toPerson();
}

operation String toPerson() : Migrated!Person {
 var person := new Migrated!Person;

 person.name := self.toName();
 person.email := self.toEmail();

 return person;
}
```

Listing C.3: The Newsgroup Extract Person model migration in Flock



(a) Original metamodel.



(b) Evolved metamodel.

Figure C.2: Newsgroups metamodel during the Resolve Replies iteration

C.1.2 Resolve Replies

The Resolve Replies iteration made explicit the lineage of each article by moving replies to an article such that they were contained in the original article. At the start of the iteration (Figure C.2(a)), each `Article` was assigned a unique identifier in the `articleId` feature. The `inReplyTo` feature was specified for `Articles` written in reply to others. At the end of the iteration, the `inReplyTo` attribute was replaced with a reference of type `Article`. The `inReplyTo` attribute was renamed to `inReplyToId` (and, in a future iteration, was removed from the metamodel).

Listings C.4, C.5 and C.6 show the model migration strategy in ATL, COPE and Flock respectively. Migration involved dereferencing the `inReplyTo` value to determine a parent `Article`, and then setting the `inReplyTo` reference to the parent `Article`.

```
module ResolveReplies;

create Migrated : After from Original : Before;

rule Newsgroups {
  from
    o : Before!Newsgroup
  to
    m : After!Newsgroup (
      articles <- o.articles
    )
}

```

```
rule Articles {
  from
    o : Before!Article
  to
    m : After!Article (
      articleId <- o.articleId,
      subject  <- o.subject,
      newsgroups <- o.newsgroups,
      inReplyToId <- o.inReplyTo,
      date     <- o.date,
      sender   <- o.sender
    )
  do {
    if (not o.inReplyTo.oclIsUndefined() and After!Article.allInstances
        ()->exists(a|a.articleId = o.inReplyTo)) {
      After!Article.allInstances()->select(a|a.articleId = o.inReplyTo)->
      first().replies <- m;
    }
  }
}
```

Listing C.4: The Newsgroup Resolve Replies model migration in ATL

```
for (article in extractperson.Article.allInstances) {
  def replyToId = article.unset(replyTo)
  article.replyToId = replyToId
  article.replyTo = Article.allInstances.find { it.articledId = article.
     replyToId }
}
```

Listing C.5: The Newsgroup Resolve Replies model migration in COPE

```
migrate Article {
  migrated.inReplyToId := original.inReplyTo;
  migrated.inReplyTo := Migrated!Article.all.selectOne(a|a.articleId =
     migrated.inReplyToId);
}
```

Listing C.6: The Newsgroup Resolve Replies model migration in Flock

C.2 UML Example

This section describes the co-evolution example taken from the evolution of the Unified Modeling Language (UML) between versions 1.4 [OMG 2001] and 2.2 [OMG 2007b]. Activity diagrams, in particular, changed radically between UML versions 1.4 and 2.2. In the former, activities were defined as a special case of state machines, while in the latter they were defined atop a more general semantic base[1] [Selic 2005].

The UML 1.4 and 2.2 specifications are defined in different metamodelling languages. The former uses XMI 1.4 and the latter XMI 2.2. Of the co-evolution tools discussed in this thesis, only Epsilon Flock can be used with both XMI 1.4 and XMI 2.2. To enable the use of other co-evolution tools with the UML metamodel changes, the author reconstructed part of the UML 1.4 metamodel in XMI 2.2.

The migration semantics were identified by comparing the UML 1.4 and UML 2.2 specifications, and by discussing the metamodel evolution with other UML experts. As described in Section 6.4, the UML 2.2 specification appears to be ambiguous with respect to the way in which UML 1.4 `ObjectFlowStates` should be migrated to conform to the UML 2.2 metamodel. The migration strategies presented here assume the semantics of the core task described in Section 6.4: `ObjectFlowStates` are replaced with `ObjectNodes`.

C.2.1 Activity Diagrams

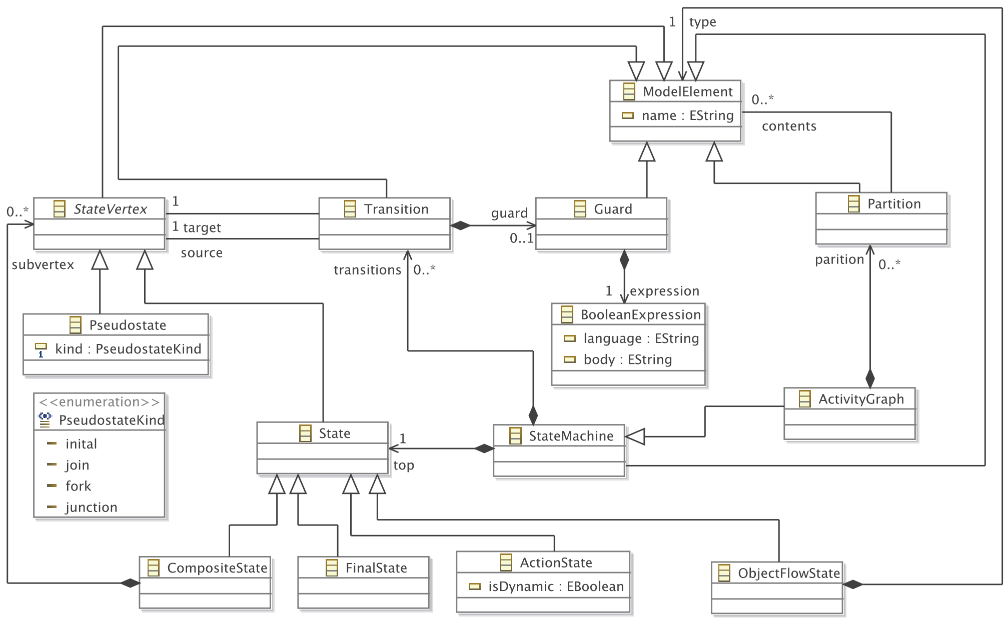
Figures C.3(a) and C.3(b) are simplifications of the activity diagram metamodels from versions 1.4 and 2.2 of the UML specification, respectively. In the interest of clarity, some features and abstract classes have been removed from Figures C.3(a) and C.3(b).

Some differences between Figures C.3(a) and C.3(b) are: activities have been changed such that they comprise nodes and edges, actions replace states in UML 2.2, and the subtypes of control node replace pseudostates.

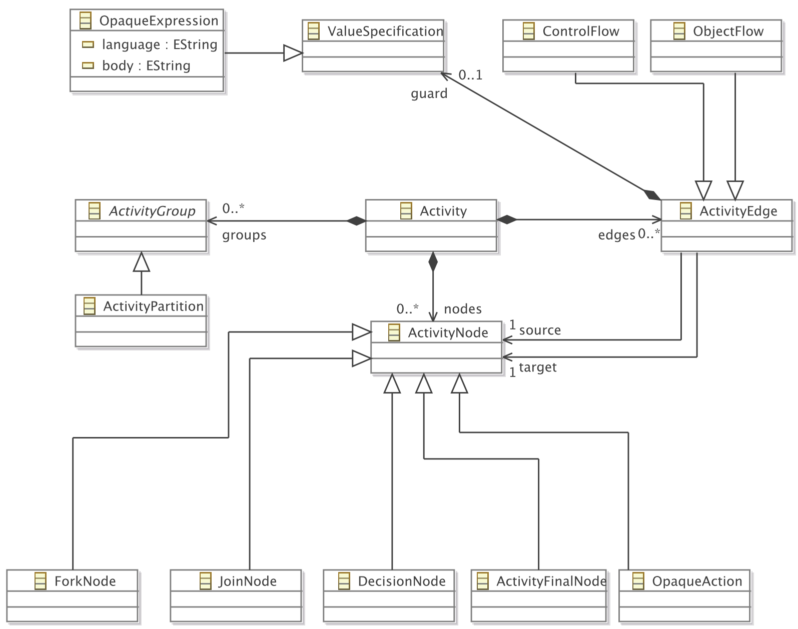
Listings C.7, C.8 and C.9 show the model migration strategy in ATL, COPE and Flock respectively. Migration mostly involved restructuring data by storing values in features of a different name, and retyping `Pseudeostates`.

```
module ActivityGraph;

create Migrated : After from Original : Before;

rule ActivityGraph {
 from
  o : Before!ActivityGraph
 to
  p : After!Package (
   packagedElement <- m
  ),
  m : After!Activity (
   name <- o.name,
   node <- o.top.subvertex,
```

[1] A variant of generalised coloured Petri nets.



(a) Original metamodel.



(b) Evolved metamodel.

Figure C.3: Activities in UML 1.4 and UML 2.2

```
      edge <- o.transitions,
      group <- o.partition
    )
}

rule Partitions {
  from
    o : Before!Partition
  to
    p : After!ActivityPartition (
      name <- o.name,
      edge <- o.contents->select(c|c.oclIsKindOf(Before!Transition)),
      node <- o.contents->reject(c|c.oclIsKindOf(Before!ObjectFlowState))
    )
}

rule ActionState2OpaqueAction {
  from
    o : Before!ActionState
  to
    p : After!OpaqueAction (
      name <- o.name
    )
}

rule Initials {
  from
    o : Before!Pseudostate (
        o.kind = #inital
      )
  to
    p : After!InitialNode
}

rule Decisions {
  from
    o : Before!Pseudostate (
        o.kind = #junction
      )
  to
    p : After!DecisionNode
}

rule Forks {
  from
    o : Before!Pseudostate (
        o.kind = #fork
```

```
    )
  to
    p : After!ForkNode
}

rule Joins {
  from
    o : Before!Pseudostate (
       o.kind = #join
    )
  to
    p : After!MergeNode
}

rule Finals {
  from
    o : Before!FinalState
  to
    p : After!ActivityFinalNode
}

rule ObjectFlows {
  from
    o : Before!Transition (
      o.target.oclIsTypeOf(Before!ObjectFlowState)
    )
  to
    p : After!ObjectFlow (
      source <- o.source,
      target <- o.target.outgoing->first().target
    )
}

rule ControlFlows {
  from
    o : Before!Transition (
      not o.source.oclIsTypeOf(Before!ObjectFlowState) and
      not o.target.oclIsTypeOf(Before!ObjectFlowState)
    )
  to
    p : After!ControlFlow (
      guard <- o.guard,
      source <- o.source,
      target <- o.target
    )
}
```

```
rule Guards {
  from
    o : Before!Guard
  to
    p : After!OpaqueExpression (
      body <- o.expression.body
    )
}
```

Listing C.7: UML activity diagram model migration in ATL

```
for (model in activities.Model.allInstances) {
  model.migrate(activities.Package)
  def ownedElement = model.unset(ownedElement)
  model.packagedElement = ownedElement
}

for (activity in activities.ActivityGraph.allInstances) {
  activity.migrate(activities.Activity)
  def top = activity.unset(top)
  activity.node = top.subvertex
  def transitions = activity.unset(transitions)
  activity.edge = transitions
  def partition = activity.unset(partition)
  activity.group = partition
}

for (partition in activities.ActivityGraph.allInstances) {
  def contents = partition.unset(contents)
  partition.edges = contents.findAll{it -> it instanceof activities.Transition}
  partition.nodes = contents.findAll{it -> it instanceof activities.StateVertex and not (it instanceof activities.ObjectFlowState)}
}

for (action in activities.ActionState.allInstances) {
  action.migrate(activities.OpaqueAction)
}

for (pseudostate in activities.Pseudeostate) {
  switch ( pseudostate.kind.toString() ) {
      case "pk_initial":
         pseudeostate.migrate(activities.InitialNode); break
    case "pk_junction"
      pseudeostate.migrate(activities.DecisionNode); break
    case "pk_fork"
      pseudeostate.migrate(activities.ForkNode); break
```

```
    case "pk_join"
     pseudeostate.migrate(activities.JoinNode); break
  }
}

for (finalstate in activities.FinalState.allInstances) {
  finalstate.migrate(activities.ActivityFinalNode)
}

for (transition in activities.ObjectFlow.allInstances.findAll{it -> it.
    target instanceof activities.ObjectFlowState}) {
  transition.target = transition.target.outgoing.first.target
}

for (transition in activities.Transition.allInstances) {
  transition.migrate(activities.ControlFlow)
}

for (guard in activities.Guard.allInstances) {
  transition.migrate(activities.OpaqueExpression)
  def expression = transition.unset(expression)
  transition.body = expression.body
}
```

Listing C.8: UML activity diagram model migration in COPE

```
migrate Model to Package {
  migrated.packagedElement := original.ownedElement.equivalent();
}

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

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

migrate ActionState to OpaqueAction

migrate Pseudostate to InitialNode when: original.kind.toString() = '
    pk_initial'
```

```
migrate Pseudostate to DecisionNode when: original.kind.toString() = 'pk_junction'
migrate Pseudostate to ForkNode when: original.kind.toString() = 'pk_fork'
migrate Pseudostate to JoinNode when: original.kind.toString() = 'pk_join'

migrate FinalState to ActivityFinalNode

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

migrate Transition to ControlFlow

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

Listing C.9: UML activity diagram model migration in Flock

C.3 GMF Examples

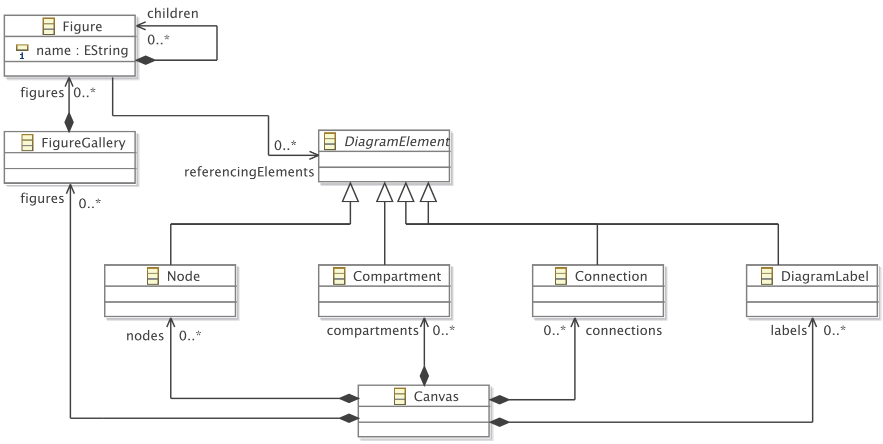
C.3.1 GMF Graph

The GMF Graph metamodel comprises approximately 60 classes. For clarity, only those classes that were affected by the changes made between versions 1.0 and 2.0 of GMF are shown in Figure C.4. The migration strategies were specified on the complete metamodel, and not only the extract shown here.

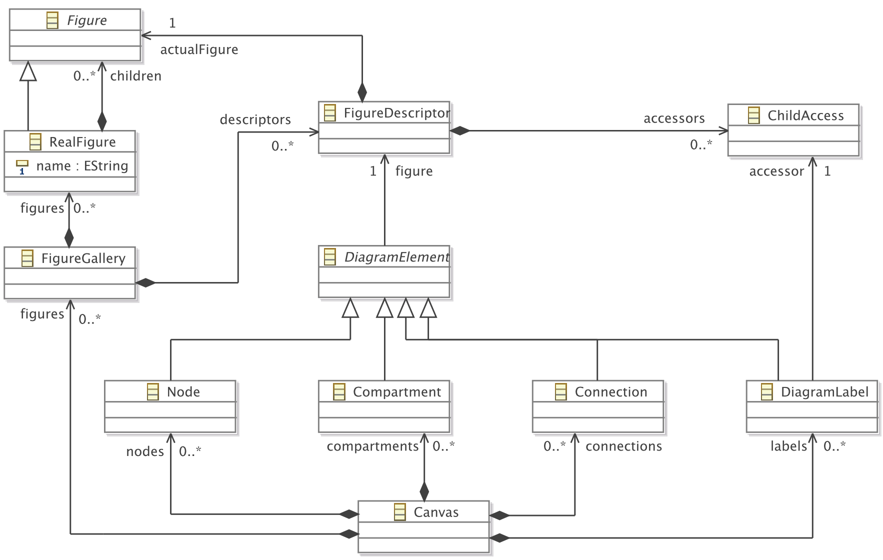
The GMF Graph metamodel (Figure C.4) describes the appearance of the generated graphical model editor. The metaclasses `Canvas`, `Figure`, `Node`, `DiagramLabel`, `Connection`, and `Compartment` are used to represent components of the graphical model editor to be generated. The evolution in the GMF Graph 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`.

During the development of GMF 2.0, the Graph metamodel from GMF 1.0 was evolved – as shown in Figure 6.16(b) – to facilitate greater re-use

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



(a) Original metamodel.



(b) Evolved metamodel.

Figure C.4: The Graph metamodel in GMF 1.0 and GMF 2.0

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

Listings C.10, C.11 and C.12 show the model migration strategy in ATL, COPE and Flock respectively. Migration involved creating proxy objects for the `FigureGallery#descriptors` and `FigureDescriptor#accessors` features, and moving values to those proxy objects.

```
module Graph;

create Migrated : After from Original : Before;

rule Canvas2Canvas extends Identity2Identity {
  from
    o : Before!Canvas
  to
    m : After!Canvas (
      figures <- o.figures,
      nodes <- o.nodes,
      connections <- o.connections,
      compartments <- o.compartments,
      labels <- o.labels
    )
}
rule FigureGallery2FigureGallery extends Identity2Identity {
  from
    o : Before!FigureGallery
  to
    m : After!FigureGallery (
       implementationBundle <- o.implementationBundle
    )
}
abstract rule Identity2Identity {
  from
    o : Before!Identity
  to
    m : After!Identity (
       name <- o.name
    )
}
abstract rule DiagramElement2DiagramElement extends Identity2Identity {
  from
    o : Before!DiagramElement
  to
    m : After!DiagramElement (
       figure <- o.figure,
       facets <- o.facets
    )
}
rule Node2Node extends DiagramElement2DiagramElement {
  from
    o : Before!Node
```

```
  to
    m : After!Node (
      resizeConstraint <- o.resizeConstraint,
      affixedParentSide <- o.affixedParentSide
    )
}
rule Connection2Connection extends DiagramElement2DiagramElement {
  from
    o : Before!Connection
  to
    m : After!Connection
}
rule Compartment2Compartment extends DiagramElement2DiagramElement {
  from
    o : Before!Compartment
  to
    m : After!Compartment (
      collapsible <- o.collapsible,
      needsTitle <- o.needsTitle
    )
}
rule DiagramLabel2DiagramLabel extends Node2Node {
  from
    o : Before!DiagramLabel
  to
    m : After!DiagramLabel (
      elementIcon <- o.elementIcon
    )
}
abstract rule VisualFacet2VisualFacet {
  from
    o : Before!VisualFacet
  to
    m : After!VisualFacet
}
rule GeneralFacet2GeneralFacet extends VisualFacet2VisualFacet {
  from
    o : Before!GeneralFacet
  to
    m : After!GeneralFacet (
      identifier <- o.identifier,
      data <- o.data
    )
}
rule AlignmentFacet2AlignmentFacet extends VisualFacet2VisualFacet {
  from
    o : Before!AlignmentFacet
```

```
 to
  m : After!AlignmentFacet (
    alignment <- o.alignment
  )
}
rule GradientFacet2GradientFacet extends VisualFacet2VisualFacet {
 from
  o : Before!GradientFacet
 to
  m : After!GradientFacet (
    direction <- o.direction
  )
}
rule LabelOffsetFacet2LabelOffsetFacet extends VisualFacet2VisualFacet {
 from
  o : Before!LabelOffsetFacet
 to
  m : After!LabelOffsetFacet (
    x <- o.x,
    y <- o.y
  )
}
rule DefaultSizeFacet2DefaultSizeFacet extends VisualFacet2VisualFacet {
 from
  o : Before!DefaultSizeFacet
 to
  m : After!DefaultSizeFacet (
    defaultSize <- o.defaultSize
  )
}
abstract rule Figure2Figure extends Layoutable2Layoutable {
 from
  o : Before!Figure
 to
  m : After!Figure (
    foregroundColor <- o.foregroundColor,
    backgroundColor <- o.backgroundColor,
    maximumSize <- o.maximumSize,
    minimumSize <- o.minimumSize,
    preferredSize <- o.preferredSize,
    font <- o.font,
    insets <- o.insets,
    border <- o.border,
    location <- o.location,
```

```
      size <- o.size
    )
}
rule FigureRef2FigureRef extends Layoutable2Layoutable {
  from
    o : Before!FigureRef
  to
    m : After!FigureRef (
      figure <- o.figure
    )
}
abstract rule Shape2Shape extends Figure2Figure {
  from
    o : Before!Shape
  to
    m : After!Shape (
      outline <- o.outline,
      fill <- o.fill,
      lineWidth <- o.lineWidth,
      lineKind <- o.lineKind,
      xorFill <- o.xorFill,
      xorOutline <- o.xorOutline,
      resolvedChildren <- o.resolvedChildren
    )
}
rule Label2Label extends Figure2Figure {
  from
    o : Before!Label
  to
    m : After!Label (
      text <- o.text
    )
}
rule LabeledContainer2LabeledContainer extends Figure2Figure {
  from
    o : Before!LabeledContainer
  to
    m : After!LabeledContainer
}
rule Rectangle2Rectangle extends Shape2Shape {
  from
    o : Before!Rectangle
  to
    m : After!Rectangle
}
rule RoundedRectangle2RoundedRectangle extends Shape2Shape {
```

```
  from
    o : Before!RoundedRectangle
  to
    m : After!RoundedRectangle (
      cornerWidth <- o.cornerWidth,
      cornerHeight <- o.cornerHeight
    )
}
rule Ellipse2Ellipse extends Shape2Shape {
  from
    o : Before!Ellipse
  to
    m : After!Ellipse
}
rule Polyline2Polyline extends Shape2Shape {
  from
    o : Before!Polyline
  to
    m : After!Polyline (
      template <- o.template
    )
}
rule Polygon2Polygon extends Polyline2Polyline {
  from
    o : Before!Polygon
  to
    m : After!Polygon
}
rule ScalablePolygon2ScalablePolygon extends Polygon2Polygon {
  from
    o : Before!ScalablePolygon
  to
    m : After!ScalablePolygon
}
rule PolylineConnection2PolylineConnection extends Polyline2Polyline {
  from
    o : Before!PolylineConnection
  to
    m : After!PolylineConnection (
      sourceDecoration <- o.sourceDecoration,
      targetDecoration <- o.targetDecoration
    )
}
rule PolylineDecoration2PolylineDecoration extends Polyline2Polyline {
  from
    o : Before!PolylineDecoration
  to
```

```
  m : After!PolylineDecoration
}
rule PolygonDecoration2PolygonDecoration extends Polygon2Polygon {
 from
  o : Before!PolygonDecoration
 to
  m : After!PolygonDecoration
}
abstract rule CustomClass2CustomClass {
 from
  o : Before!CustomClass
 to
  m : After!CustomClass (
    qualifiedClassName <- o.qualifiedClassName,
    attributes <- o.attributes
  )
}
rule CustomAttribute2CustomAttribute {
 from
  o : Before!CustomAttribute
 to
  m : After!CustomAttribute (
    name <- o.name,
    value <- o.value,
    directAccess <- o.directAccess,
    multiStatementValue <- o.multiStatementValue
  )
}
rule FigureAccessor2FigureAccessor {
 from
  o : Before!FigureAccessor
 to
  m : After!FigureAccessor (
    accessor <- o.accessor,
    typedFigure <- o.typedFigure
  )
}
rule CustomFigure2CustomFigure extends Figure2Figure {
 from
  o : Before!CustomFigure
 to
  m : After!CustomFigure (
    customChildren <- o.customChildren
  )
}
rule CustomDecoration2CustomDecoration extends
```

```
    CustomFigure2CustomFigure {
  from
    o : Before!CustomDecoration
  to
    m : After!CustomDecoration
}
rule CustomConnection2CustomConnection extends
    CustomFigure2CustomFigure {
  from
    o : Before!CustomConnection
  to
    m : After!CustomConnection
}
abstract rule Color2Color {
  from
    o : Before!Color
  to
    m : After!Color
}
rule RGBColor2RGBColor extends Color2Color {
  from
    o : Before!RGBColor
  to
    m : After!RGBColor (
      red <- o.red,
      green <- o.green,
      blue <- o.blue
    )
}
rule ConstantColor2ConstantColor extends Color2Color {
  from
    o : Before!ConstantColor
  to
    m : After!ConstantColor (
      value <- o.value
    )
}
abstract rule Font2Font {
  from
    o : Before!Font
  to
    m : After!Font
}
rule BasicFont2BasicFont extends Font2Font {
  from
    o : Before!BasicFont
  to
```

```
    m : After!BasicFont (
      faceName <- o.faceName,
      height <- o.height,
      style <- o.style
    )
}
rule Point2Point {
  from
    o : Before!Point
  to
    m : After!Point (
      x <- o.x,
      y <- o.y
    )
}
rule Dimension2Dimension {
  from
    o : Before!Dimension
  to
    m : After!Dimension (
      dx <- o.dx,
      dy <- o.dy
    )
}
rule Insets2Insets {
  from
    o : Before!Insets
  to
    m : After!Insets (
      top <- o.top,
      left <- o.left,
      bottom <- o.bottom,
      right <- o.right
    )
}
abstract rule Border2Border {
  from
    o : Before!Border
  to
    m : After!Border
}
rule LineBorder2LineBorder extends Border2Border {
  from
    o : Before!LineBorder
  to
    m : After!LineBorder (
```

```
      color <- o.color,
      width <- o.width
    )
}
rule MarginBorder2MarginBorder extends Border2Border {
  from
    o : Before!MarginBorder
  to
    m : After!MarginBorder (
      insets <- o.insets
    )
}
rule CompoundBorder2CompoundBorder extends Border2Border {
  from
    o : Before!CompoundBorder
  to
    m : After!CompoundBorder (
      outer <- o.outer,
      inner <- o.inner
    )
}
rule CustomBorder2CustomBorder extends Border2Border {
  from
    o : Before!CustomBorder
  to
    m : After!CustomBorder
}
abstract rule LayoutData2LayoutData {
  from
    o : Before!LayoutData
  to
    m : After!LayoutData (
      owner <- o.owner
    )
}
rule CustomLayoutData2CustomLayoutData extends LayoutData2LayoutData {
  from
    o : Before!CustomLayoutData
  to
    m : After!CustomLayoutData
}
rule GridLayoutData2GridLayoutData extends LayoutData2LayoutData {
  from
    o : Before!GridLayoutData
  to
    m : After!GridLayoutData (
      grabExcessHorizontalSpace <- o.grabExcessHorizontalSpace,
```

```
      grabExcessVerticalSpace <- o.grabExcessVerticalSpace,
      verticalAlignment <- o.verticalAlignment,
      horizontalAlignment <- o.horizontalAlignment,
      verticalSpan <- o.verticalSpan,
      horizontalSpan <- o.horizontalSpan,
      horizontalIndent <- o.horizontalIndent,
      sizeHint <- o.sizeHint
    )
}
rule BorderLayoutData2BorderLayoutData extends LayoutData2LayoutData {
  from
    o : Before!BorderLayoutData
  to
    m : After!BorderLayoutData (
      alignment <- o.alignment,
      vertical <- o.vertical
    )
}
abstract rule Layoutable2Layoutable {
  from
    o : Before!Layoutable
  to
    m : After!Layoutable (
      layoutData <- o.layoutData,
      layout <- o.layout
    )
}
abstract rule Layout2Layout {
  from
    o : Before!Layout
  to
    m : After!Layout
}
rule CustomLayout2CustomLayout extends Layout2Layout {
  from
    o : Before!CustomLayout
  to
    m : After!CustomLayout
}
rule GridLayout2GridLayout extends Layout2Layout {
  from
    o : Before!GridLayout
  to
    m : After!GridLayout (
      numColumns <- o.numColumns,
      equalWidth <- o.equalWidth,
```

```
      margins <- o.margins,
      spacing <- o.spacing
    )
}
rule BorderLayout2BorderLayout extends Layout2Layout {
  from
    o : Before!BorderLayout
  to
    m : After!BorderLayout (
      spacing <- o.spacing
    )
}
rule FlowLayout2FlowLayout extends Layout2Layout {
  from
    o : Before!FlowLayout
  to
    m : After!FlowLayout (
      vertical <- o.vertical,
      matchMinorSize <- o.matchMinorSize,
      forceSingleLine <- o.forceSingleLine,
      majorAlignment <- o.majorAlignment,
      minorAlignment <- o.minorAlignment,
      majorSpacing <- o.majorSpacing,
      minorSpacing <- o.minorSpacing
    )
}
rule XYLayout2XYLayout extends Layout2Layout {
  from
    o : Before!XYLayout
  to
    m : After!XYLayout
}
rule XYLayoutData2XYLayoutData extends LayoutData2LayoutData {
  from
    o : Before!XYLayoutData
  to
    m : After!XYLayoutData (
      topLeft <- o.topLeft,
      size <- o.size
    )
}
rule StackLayout2StackLayout extends Layout2Layout {
  from
    o : Before!StackLayout
  to
    m : After!StackLayout
```

```
}
```

Listing C.10: GMF Graph model migration in ATL

```
for (gallery in graph.FigureGallery.allInstances) {
 while(not gallery.figures.isEmpty()) {
  def figure = gallery.figures.first()
  def descriptor = graph.FigureDescriptor.newInstance()

  descriptor.name = figure.name
  descriptor.actualFigure = figure

  figure.set(descriptor, descriptor)

  figure.children.findAll{ it -> it instanceof graph.Label}.each do |
      it|
   def accessor = graph.ChildAccess.newInstance()

   accessor.figure = it
   descriptor.accessors.add(accessor)

   it.set(accessor, accessor)
  end

  return descriptor;
 }
}

for (diagramElement in graph.DiagramElement.allInstances()) {
  diagramElement.figure.unset(descriptor)
 diagramElement.figure = descriptor
}

for (diagramLabel in graph.DiagramLabel.allInstances()) {
  diagramElement.figure.unset(accessor)
 diagramElement.accessor = accessor
}
```

Listing C.11: GMF Graph model migration in COPE

```
migrate FigureGallery {
 while (not migrated.figures.isEmpty()) {
  migrated.descriptors.add(migrated.figures.first.createDescriptor());
 }
}

migrate Compartment {
```

```
  migrated.figure := original.figure.equivalent().~descriptor;
}

migrate Connection {
  migrated.figure := original.figure.equivalent().~descriptor;
}

migrate DiagramLabel {
  migrated.figure := original.figure.equivalent().~descriptor;
  migrated.accessor := original.figure.equivalent().~accessor;
}

migrate Node {
  migrated.figure := original.figure.equivalent().~descriptor;
}

operation Migrated!Figure createDescriptor() : Migrated!
    FigureDescriptor {
  var descriptor := new Migrated!FigureDescriptor;

  descriptor.name := self.name;
  descriptor.actualFigure := self;

  self.~descriptor := descriptor;

  self.children.forAll(l : Migrated!Label | l.addAccessor(descriptor));

  return descriptor;
}

operation Migrated!Label addAccessor(descriptor : Migrated!
    FigureDescriptor) {
  var accessor := new Migrated!ChildAccess;

  accessor.figure := self;
  self.~descriptor := descriptor;
  self.~accessor := accessor;
  descriptor.accessors.add(accessor);
}
```

Listing C.12: GMF Graph model migration in Flock

C.3.2 GMF Generator

During the development of GMF vX, the Generator metamodel evolved to make explicit the use of `ContextMenus` and `Parsers`. In previous versions of GMF, `ContextMenus` and `Parsers` were not customisable via the Generator metamodel. Instead, the GMF runtime created menus and parsers automatically at runtime. The GMF generator metamodel is too large to show here, comprising approximately 150 classes. The changes made between versions X and X of GMF affected at least 20 of those classes.

Listings C.13, C.13 and C.13 show the model migration strategy in ATL, COPE and Flock respectively. Migration involved populating `ContextMenus` from existing diagram elements, and creating `Parsers` for built-in and user-defined languages.

```
module GenModel2009;

create Migrated : After from Original : Before;

rule GenEditorGenerator2GenEditorGenerator {
  from
    o : Before!GenEditorGenerator
  to
    m : After!GenEditorGenerator (
      audits <- o.audits,
      metrics <- o.metrics,
      diagram <- o.diagram,
      plugin <- o.plugin,
      editor <- o.editor,
      navigator <- o.navigator,
      diagramUpdater <- o.diagramUpdater,
      propertySheet <- o.propertySheet,
      application <- o.application,
      domainGenModel <- o.domainGenModel,
      packageNamePrefix <- o.packageNamePrefix,
      modelID <- o.modelID,
      sameFileForDiagramAndModel <- o.sameFileForDiagramAndModel,
      diagramFileExtension <- o.diagramFileExtension,
      domainFileExtension <- o.domainFileExtension,
      dynamicTemplates <- o.dynamicTemplates,
      templateDirectory <- o.templateDirectory,
      copyrightText <- o.copyrightText,
      expressionProviders <- o.expressionProviders,
      modelAccess <- o.modelAccess
    )
}
rule GenDiagram2GenDiagram extends GenContainerBase2GenContainerBase {
  from
    o : Before!GenDiagram
  to
```

```
    m : After!GenDiagram (
      domainDiagramElement <- o.domainDiagramElement,
      childNodes <- o.childNodes,
      topLevelNodes <- o.topLevelNodes,
      links <- o.links,
      compartments <- o.compartments,
      palette <- o.palette,
      synchronized <- o.synchronized,
      preferences <- o.preferences,
      preferencePages <- o.preferencePages
    )
}
rule GenEditorView2GenEditorView {
  from
    o : Before!GenEditorView
  to
    m : After!GenEditorView (
      packageName <- o.packageName,
      actionBarContributorClassName <- o.actionBarContributorClassName,
      className <- o.className,
      iconPath <- o.iconPath,
      iD <- o.iD,
      eclipseEditor <- o.eclipseEditor,
      contextID <- o.contextID
    )
}
abstract rule GenPreferencePage2GenPreferencePage {
  from
    o : Before!GenPreferencePage
  to
    m : After!GenPreferencePage (
      iD <- o.iD,
      name <- o.name,
      children <- o.children
    )
}
rule GenCustomPreferencePage2GenCustomPreferencePage extends GenPreferencePage2GenPreferencePage {
  from
    o : Before!GenCustomPreferencePage
  to
    m : After!GenCustomPreferencePage (
      qualifiedClassName <- o.qualifiedClassName
    )
}
```

```
rule GenStandardPreferencePage2GenStandardPreferencePage extends
    GenPreferencePage2GenPreferencePage {
 from
  o : Before!GenStandardPreferencePage
 to
  m : After!GenStandardPreferencePage (
   kind <- o.kind
  )
}
rule GenDiagramPreferences2GenDiagramPreferences {
 from
  o : Before!GenDiagramPreferences
 to
  m : After!GenDiagramPreferences (
   lineStyle <- o.lineStyle,
   defaultFont <- o.defaultFont,
   fontColor <- o.fontColor,
   fillColor <- o.fillColor,
   lineColor <- o.lineColor,
   noteFillColor <- o.noteFillColor,
   noteLineColor <- o.noteLineColor,
   showConnectionHandles <- o.showConnectionHandles,
   showPopupBars <- o.showPopupBars,
   promptOnDelFromModel <- o.promptOnDelFromModel,
   promptOnDelFromDiagram <- o.promptOnDelFromDiagram,
   enableAnimatedLayout <- o.enableAnimatedLayout,
   enableAnimatedZoom <- o.enableAnimatedZoom,
   enableAntiAlias <- o.enableAntiAlias,
   showGrid <- o.showGrid,
   showRulers <- o.showRulers,
   snapToGrid <- o.snapToGrid,
   snapToGeometry <- o.snapToGeometry,
   gridInFront <- o.gridInFront,
   rulerUnits <- o.rulerUnits,
   gridSpacing <- o.gridSpacing,
   gridLineColor <- o.gridLineColor,
   gridLineStyle <- o.gridLineStyle
  )
}
abstract rule GenFont2GenFont {
 from
  o : Before!GenFont
 to
  m : After!GenFont
}
```

```
rule GenStandardFont2GenStandardFont extends GenFont2GenFont {
  from
    o : Before!GenStandardFont
  to
    m : After!GenStandardFont (
      name <- o.name
    )
}
rule GenCustomFont2GenCustomFont extends GenFont2GenFont {
  from
    o : Before!GenCustomFont
  to
    m : After!GenCustomFont (
      name <- o.name,
      height <- o.height,
      style <- o.style
    )
}
abstract rule GenColor2GenColor {
  from
    o : Before!GenColor
  to
    m : After!GenColor
}
rule GenRGBColor2GenRGBColor extends GenColor2GenColor {
  from
    o : Before!GenRGBColor
  to
    m : After!GenRGBColor (
      red <- o.red,
      green <- o.green,
      blue <- o.blue
    )
}
rule GenConstantColor2GenConstantColor extends GenColor2GenColor {
  from
    o : Before!GenConstantColor
  to
    m : After!GenConstantColor (
      name <- o.name
    )
}
rule GenDiagramUpdater2GenDiagramUpdater {
  from
    o : Before!GenDiagramUpdater
  to
    m : After!GenDiagramUpdater (
```

```
      diagramUpdaterClassName <- o.diagramUpdaterClassName,
      nodeDescriptorClassName <- o.nodeDescriptorClassName,
      linkDescriptorClassName <- o.linkDescriptorClassName,
      updateCommandClassName <- o.updateCommandClassName,
      updateCommandID <- o.updateCommandID
    )
}
rule GenPlugin2GenPlugin {
  from
    o : Before!GenPlugin
  to
    m : After!GenPlugin (
      iD <- o.iD,
      name <- o.name,
      provider <- o.provider,
      version <- o.version,
      printingEnabled <- o.printingEnabled,
      requiredPlugins <- o.requiredPlugins,
      activatorClassName <- o.activatorClassName
    )
}
rule DynamicModelAccess2DynamicModelAccess {
  from
    o : Before!DynamicModelAccess
  to
    m : After!DynamicModelAccess (
      packageName <- o.packageName,
      className <- o.className
    )
}
abstract rule GenCommonBase2GenCommonBase {
  from
    o : Before!GenCommonBase
  to
    m : After!GenCommonBase (
      diagramRunTimeClass <- o.diagramRunTimeClass,
      visualID <- o.visualID,
      elementType <- o.elementType,
      editPartClassName <- o.editPartClassName,
      itemSemanticEditPolicyClassName <- o.
      itemSemanticEditPolicyClassName,
      notationViewFactoryClassName <- o.notationViewFactoryClassName,
      viewmap <- o.viewmap,
      styles <- o.styles,
      behaviour <- o.behaviour
```

```
    )
}
abstract rule Behaviour2Behaviour {
  from
    o : Before!Behaviour
  to
    m : After!Behaviour
}
rule CustomBehaviour2CustomBehaviour extends Behaviour2Behaviour {
  from
    o : Before!CustomBehaviour
  to
    m : After!CustomBehaviour (
      key <- o.key,
      editPolicyQualifiedClassName <- o.editPolicyQualifiedClassName
    )
}
rule SharedBehaviour2SharedBehaviour extends Behaviour2Behaviour {
  from
    o : Before!SharedBehaviour
  to
    m : After!SharedBehaviour (
      delegate <- o.delegate
    )
}
rule OpenDiagramBehaviour2OpenDiagramBehaviour extends
     Behaviour2Behaviour {
  from
    o : Before!OpenDiagramBehaviour
  to
    m : After!OpenDiagramBehaviour (
      editPolicyClassName <- o.editPolicyClassName,
      diagramKind <- o.diagramKind,
      editorID <- o.editorID,
      openAsEclipseEditor <- o.openAsEclipseEditor
    )
}
abstract rule GenContainerBase2GenContainerBase extends
     GenCommonBase2GenCommonBase {
  from
    o : Before!GenContainerBase
  to
    m : After!GenContainerBase (
      canonicalEditPolicyClassName <- o.canonicalEditPolicyClassName
    )
}
abstract rule GenChildContainer2GenChildContainer extends
```

```
    GenContainerBase2GenContainerBase {
  from
    o : Before!GenChildContainer
  to
    m : After!GenChildContainer (
      childNodes <- o.childNodes
    )
}
abstract rule GenNode2GenNode extends
    GenChildContainer2GenChildContainer {
  from
    o : Before!GenNode
  to
    m : After!GenNode (
      modelFacet <- o.modelFacet,
      labels <- o.labels,
      compartments <- o.compartments,
      primaryDragEditPolicyQualifiedClassName <- o.
    primaryDragEditPolicyQualifiedClassName,
      graphicalNodeEditPolicyClassName <- o.
    graphicalNodeEditPolicyClassName,
      createCommandClassName <- o.createCommandClassName
    )
}
rule GenTopLevelNode2GenTopLevelNode extends GenNode2GenNode {
  from
    o : Before!GenTopLevelNode
  to
    m : After!GenTopLevelNode
}
rule GenChildNode2GenChildNode extends GenNode2GenNode {
  from
    o : Before!GenChildNode
  to
    m : After!GenChildNode
}
rule GenChildSideAffixedNode2GenChildSideAffixedNode extends
    GenChildNode2GenChildNode {
  from
    o : Before!GenChildSideAffixedNode
  to
    m : After!GenChildSideAffixedNode (
      preferredSideName <- o.preferredSideName
    )
}
rule GenChildLabelNode2GenChildLabelNode extends
    GenChildNode2GenChildNode {
```

```
  from
    o : Before!GenChildLabelNode
  to
    m : After!GenChildLabelNode (
      labelReadOnly <- o.labelReadOnly,
      labelElementIcon <- o.labelElementIcon,
      labelModelFacet <- o.labelModelFacet
    )
}
rule GenCompartment2GenCompartment extends
    GenChildContainer2GenChildContainer {
  from
    o : Before!GenCompartment
  to
    m : After!GenCompartment (
      title <- o.title,
      canCollapse <- o.canCollapse,
      hideIfEmpty <- o.hideIfEmpty,
      needsTitle <- o.needsTitle,
      node <- o.node,
      listLayout <- o.listLayout
    )
}
rule GenLink2GenLink extends GenCommonBase2GenCommonBase {
  from
    o : Before!GenLink
  to
    m : After!GenLink (
      modelFacet <- o.modelFacet,
      labels <- o.labels,
      outgoingCreationAllowed <- o.outgoingCreationAllowed,
      incomingCreationAllowed <- o.incomingCreationAllowed,
      viewDirectionAlignedWithModel <- o.viewDirectionAlignedWithModel,
      creationConstraints <- o.creationConstraints,
      createCommandClassName <- o.createCommandClassName,
      reorientCommandClassName <- o.reorientCommandClassName,
      treeBranch <- o.treeBranch
    )
}
abstract rule GenLabel2GenLabel extends GenCommonBase2GenCommonBase {
  from
    o : Before!GenLabel
  to
    m : After!GenLabel (
      readOnly <- o.readOnly,
```

```
      elementIcon <- o.elementIcon,
      modelFacet <- o.modelFacet
    )
}
rule GenNodeLabel2GenNodeLabel extends GenLabel2GenLabel {
  from
    o : Before!GenNodeLabel
  to
    m : After!GenNodeLabel
}
rule GenExternalNodeLabel2GenExternalNodeLabel extends
    GenNodeLabel2GenNodeLabel {
  from
    o : Before!GenExternalNodeLabel
  to
    m : After!GenExternalNodeLabel
}
rule GenLinkLabel2GenLinkLabel extends GenLabel2GenLabel {
  from
    o : Before!GenLinkLabel
  to
    m : After!GenLinkLabel (
      link <- o.link,
      alignment <- o.alignment
    )
}
abstract rule ElementType2ElementType {
  from
    o : Before!ElementType
  to
    m : After!ElementType (
      diagramElement <- o.diagramElement,
      uniqueIdentifier <- o.uniqueIdentifier,
      displayName <- o.displayName,
      definedExternally <- o.definedExternally
    )
}
rule MetamodelType2MetamodelType extends ElementType2ElementType {
  from
    o : Before!MetamodelType
  to
    m : After!MetamodelType (
      editHelperClassName <- o.editHelperClassName
    )
}
rule SpecializationType2SpecializationType extends
```

```
    ElementType2ElementType {
  from
    o : Before!SpecializationType
  to
    m : After!SpecializationType (
      metamodelType <- o.metamodelType,
      editHelperAdviceClassName <- o.editHelperAdviceClassName
    )
}
rule NotationType2NotationType extends ElementType2ElementType {
  from
    o : Before!NotationType
  to
    m : After!NotationType
}
abstract rule ModelFacet2ModelFacet {
  from
    o : Before!ModelFacet
  to
    m : After!ModelFacet
}
abstract rule LinkModelFacet2LinkModelFacet extends
    ModelFacet2ModelFacet {
  from
    o : Before!LinkModelFacet
  to
    m : After!LinkModelFacet
}
abstract rule LabelModelFacet2LabelModelFacet extends
    ModelFacet2ModelFacet {
  from
    o : Before!LabelModelFacet
  to
    m : After!LabelModelFacet
}
rule TypeModelFacet2TypeModelFacet extends ModelFacet2ModelFacet {
  from
    o : Before!TypeModelFacet
  to
    m : After!TypeModelFacet (
      metaClass <- o.metaClass,
      containmentMetaFeature <- o.containmentMetaFeature,
      childMetaFeature <- o.childMetaFeature,
      modelElementSelector <- o.modelElementSelector,
      modelElementInitializer <- o.modelElementInitializer
    )
}
```

```
rule TypeLinkModelFacet2TypeLinkModelFacet extends
    TypeModelFacet2TypeModelFacet {
  from
    o : Before!TypeLinkModelFacet
  to
    m : After!TypeLinkModelFacet (
      sourceMetaFeature <- o.sourceMetaFeature,
      targetMetaFeature <- o.targetMetaFeature
    )
}
rule FeatureLinkModelFacet2FeatureLinkModelFacet extends
    LinkModelFacet2LinkModelFacet {
  from
    o : Before!FeatureLinkModelFacet
  to
    m : After!FeatureLinkModelFacet (
      metaFeature <- o.metaFeature
    )
}
rule FeatureLabelModelFacet2FeatureLabelModelFacet extends
    LabelModelFacet2LabelModelFacet {
  from
    o : Before!FeatureLabelModelFacet
  to
    m : After!FeatureLabelModelFacet (
      metaFeatures <- o.metaFeatures,
      viewPattern <- o.viewPattern,
      editorPattern <- o.editorPattern,
      editPattern <- o.editPattern,
      viewMethod <- o.viewMethod,
      editMethod <- o.editMethod
    )
}
rule DesignLabelModelFacet2DesignLabelModelFacet extends
    LabelModelFacet2LabelModelFacet {
  from
    o : Before!DesignLabelModelFacet
  to
    m : After!DesignLabelModelFacet
}
abstract rule Attributes2Attributes {
  from
    o : Before!Attributes
  to
    m : After!Attributes
}
```

```
rule ColorAttributes2ColorAttributes extends Attributes2Attributes {
  from
    o : Before!ColorAttributes
  to
    m : After!ColorAttributes (
      foregroundColor <- o.foregroundColor,
      backgroundColor <- o.backgroundColor
    )
}
rule StyleAttributes2StyleAttributes extends Attributes2Attributes {
  from
    o : Before!StyleAttributes
  to
    m : After!StyleAttributes (
      fixedFont <- o.fixedFont,
      fixedForeground <- o.fixedForeground,
      fixedBackground <- o.fixedBackground
    )
}
rule ResizeConstraints2ResizeConstraints extends Attributes2Attributes
    {
  from
    o : Before!ResizeConstraints
  to
    m : After!ResizeConstraints (
      resizeHandles <- o.resizeHandles,
      nonResizeHandles <- o.nonResizeHandles
    )
}
rule DefaultSizeAttributes2DefaultSizeAttributes extends
    Attributes2Attributes {
  from
    o : Before!DefaultSizeAttributes
  to
    m : After!DefaultSizeAttributes (
      width <- o.width,
      height <- o.height
    )
}
rule LabelOffsetAttributes2LabelOffsetAttributes extends
    Attributes2Attributes {
  from
    o : Before!LabelOffsetAttributes
  to
    m : After!LabelOffsetAttributes (
      x <- o.x,
```

```
    y <- o.y
  )
}
abstract rule Viewmap2Viewmap {
 from
  o : Before!Viewmap
 to
  m : After!Viewmap (
    attributes <- o.attributes,
    requiredPluginIDs <- o.requiredPluginIDs,
    layoutType <- o.layoutType
  )
}
rule FigureViewmap2FigureViewmap extends Viewmap2Viewmap {
 from
  o : Before!FigureViewmap
 to
  m : After!FigureViewmap (
    figureQualifiedClassName <- o.figureQualifiedClassName
  )
}
rule SnippetViewmap2SnippetViewmap extends Viewmap2Viewmap {
 from
  o : Before!SnippetViewmap
 to
  m : After!SnippetViewmap (
    body <- o.body
  )
}
rule InnerClassViewmap2InnerClassViewmap extends Viewmap2Viewmap {
 from
  o : Before!InnerClassViewmap
 to
  m : After!InnerClassViewmap (
    className <- o.className,
    classBody <- o.classBody
  )
}
rule ParentAssignedViewmap2ParentAssignedViewmap extends
    Viewmap2Viewmap {
 from
  o : Before!ParentAssignedViewmap
 to
  m : After!ParentAssignedViewmap (
    getterName <- o.getterName,
    setterName <- o.setterName,
```

```
      figureQualifiedClassName <- o.figureQualifiedClassName
    )
}
rule ValueExpression2ValueExpression {
  from
    o : Before!ValueExpression
  to
    m : After!ValueExpression (
      body <- o.body
    )
}
rule GenConstraint2GenConstraint extends
     ValueExpression2ValueExpression {
  from
    o : Before!GenConstraint
  to
    m : After!GenConstraint
}
rule Palette2Palette {
  from
    o : Before!Palette
  to
    m : After!Palette (
      flyout <- o.flyout,
      groups <- o.groups,
      packageName <- o.packageName,
      factoryClassName <- o.factoryClassName
    )
}
abstract rule EntryBase2EntryBase {
  from
    o : Before!EntryBase
  to
    m : After!EntryBase (
      title <- o.title,
      description <- o.description,
      largeIconPath <- o.largeIconPath,
      smallIconPath <- o.smallIconPath,
      createMethodName <- o.createMethodName
    )
}
abstract rule AbstractToolEntry2AbstractToolEntry extends
     EntryBase2EntryBase {
  from
    o : Before!AbstractToolEntry
  to
```

```
    m : After!AbstractToolEntry (
      default <- o.default,
      qualifiedToolName <- o.qualifiedToolName,
      properties <- o.properties
    )
}
rule ToolEntry2ToolEntry extends AbstractToolEntry2AbstractToolEntry {
  from
    o : Before!ToolEntry
  to
    m : After!ToolEntry (
      genNodes <- o.genNodes,
      genLinks <- o.genLinks
    )
}
rule StandardEntry2StandardEntry extends
     AbstractToolEntry2AbstractToolEntry {
  from
    o : Before!StandardEntry
  to
    m : After!StandardEntry (
      kind <- o.kind
    )
}
abstract rule ToolGroupItem2ToolGroupItem {
  from
    o : Before!ToolGroupItem
  to
    m : After!ToolGroupItem
}
rule Separator2Separator extends ToolGroupItem2ToolGroupItem {
  from
    o : Before!Separator
  to
    m : After!Separator
}
rule ToolGroup2ToolGroup extends EntryBase2EntryBase {
  from
    o : Before!ToolGroup
  to
    m : After!ToolGroup (
      palette <- o.palette,
      stack <- o.stack,
      collapse <- o.collapse,
      entries <- o.entries
    )
```

```
}
abstract rule GenElementInitializer2GenElementInitializer {
  from
    o : Before!GenElementInitializer
  to
    m : After!GenElementInitializer
}
rule GenFeatureSeqInitializer2GenFeatureSeqInitializer extends
    GenElementInitializer2GenElementInitializer {
  from
    o : Before!GenFeatureSeqInitializer
  to
    m : After!GenFeatureSeqInitializer (
      initializers <- o.initializers,
      elementClass <- o.elementClass
    )
}
rule GenFeatureValueSpec2GenFeatureValueSpec extends
    GenFeatureInitializer2GenFeatureInitializer {
  from
    o : Before!GenFeatureValueSpec
  to
    m : After!GenFeatureValueSpec (
      value <- o.value
    )
}
rule GenReferenceNewElementSpec2GenReferenceNewElementSpec extends
    GenFeatureInitializer2GenFeatureInitializer {
  from
    o : Before!GenReferenceNewElementSpec
  to
    m : After!GenReferenceNewElementSpec (
      newElementInitializers <- o.newElementInitializers
    )
}
abstract rule GenFeatureInitializer2GenFeatureInitializer {
  from
    o : Before!GenFeatureInitializer
  to
    m : After!GenFeatureInitializer (
      feature <- o.feature
    )
}
rule GenLinkConstraints2GenLinkConstraints {
  from
    o : Before!GenLinkConstraints
  to
```

```
    m : After!GenLinkConstraints (
      link <- o.link,
      sourceEnd <- o.sourceEnd,
      targetEnd <- o.targetEnd
    )
}
rule GenAuditRoot2GenAuditRoot {
  from
    o : Before!GenAuditRoot
  to
    m : After!GenAuditRoot (
      categories <- o.categories,
      rules <- o.rules,
      clientContexts <- o.clientContexts
    )
}
rule GenAuditContainer2GenAuditContainer {
  from
    o : Before!GenAuditContainer
  to
    m : After!GenAuditContainer (
      id <- o.id,
      name <- o.name,
      description <- o.description,
      path <- o.path,
      audits <- o.audits
    )
}
abstract rule GenRuleBase2GenRuleBase {
  from
    o : Before!GenRuleBase
  to
    m : After!GenRuleBase (
      name <- o.name,
      description <- o.description
    )
}
rule GenAuditRule2GenAuditRule extends GenRuleBase2GenRuleBase {
  from
    o : Before!GenAuditRule
  to
    m : After!GenAuditRule (
      id <- o.id,
      rule <- o.rule,
      target <- o.target,
      message <- o.message,
```

```
      severity <- o.severity,
      useInLiveMode <- o.useInLiveMode,
      category <- o.category
    )
}
abstract rule GenRuleTarget2GenRuleTarget {
  from
    o : Before!GenRuleTarget
  to
    m : After!GenRuleTarget
}
rule GenDomainElementTarget2GenDomainElementTarget extends
    GenAuditable2GenAuditable {
  from
    o : Before!GenDomainElementTarget
  to
    m : After!GenDomainElementTarget (
      element <- o.element
    )
}
rule GenDiagramElementTarget2GenDiagramElementTarget extends
    GenAuditable2GenAuditable {
  from
    o : Before!GenDiagramElementTarget
  to
    m : After!GenDiagramElementTarget (
      element <- o.element
    )
}
rule GenDomainAttributeTarget2GenDomainAttributeTarget extends
    GenAuditable2GenAuditable {
  from
    o : Before!GenDomainAttributeTarget
  to
    m : After!GenDomainAttributeTarget (
      attribute <- o.attribute,
      nullAsError <- o.nullAsError
    )
}
rule GenNotationElementTarget2GenNotationElementTarget extends
    GenAuditable2GenAuditable {
  from
    o : Before!GenNotationElementTarget
  to
    m : After!GenNotationElementTarget (
      element <- o.element
    )
```

```
}
rule GenMetricContainer2GenMetricContainer {
  from
    o : Before!GenMetricContainer
  to
    m : After!GenMetricContainer (
      metrics <- o.metrics
    )
}
rule GenMetricRule2GenMetricRule extends GenRuleBase2GenRuleBase {
  from
    o : Before!GenMetricRule
  to
    m : After!GenMetricRule (
      key <- o.key,
      rule <- o.rule,
      target <- o.target,
      lowLimit <- o.lowLimit,
      highLimit <- o.highLimit,
      container <- o.container
    )
}
rule GenAuditedMetricTarget2GenAuditedMetricTarget extends
    GenAuditable2GenAuditable {
  from
    o : Before!GenAuditedMetricTarget
  to
    m : After!GenAuditedMetricTarget (
      metric <- o.metric,
      metricValueContext <- o.metricValueContext
    )
}
abstract rule GenAuditable2GenAuditable extends
    GenRuleTarget2GenRuleTarget {
  from
    o : Before!GenAuditable
  to
    m : After!GenAuditable (
      contextSelector <- o.contextSelector
    )
}
rule GenAuditContext2GenAuditContext {
  from
    o : Before!GenAuditContext
  to
    m : After!GenAuditContext (
```

```
      root <- o.root,
      id <- o.id,
      className <- o.className,
      ruleTargets <- o.ruleTargets
    )
}
abstract rule GenMeasurable2GenMeasurable extends
     GenRuleTarget2GenRuleTarget {
  from
    o : Before!GenMeasurable
  to
    m : After!GenMeasurable
}
rule GenExpressionProviderContainer2GenExpressionProviderContainer {
  from
    o : Before!GenExpressionProviderContainer
  to
    m : After!GenExpressionProviderContainer (
      expressionsPackageName <- o.expressionsPackageName,
      abstractExpressionClassName <- o.abstractExpressionClassName,
      providers <- o.providers
    )
}
abstract rule GenExpressionProviderBase2GenExpressionProviderBase {
  from
    o : Before!GenExpressionProviderBase
  to
    m : After!GenExpressionProviderBase (
      expressions <- o.expressions
    )
}
rule GenJavaExpressionProvider2GenJavaExpressionProvider extends
     GenExpressionProviderBase2GenExpressionProviderBase {
  from
    o : Before!GenJavaExpressionProvider
  to
    m : After!GenJavaExpressionProvider (
      throwException <- o.throwException,
      injectExpressionBody <- o.injectExpressionBody
    )
}
rule GenExpressionInterpreter2GenExpressionInterpreter extends
     GenExpressionProviderBase2GenExpressionProviderBase {
  from
    o : Before!GenExpressionInterpreter
  to
```

```
    m : After!GenExpressionInterpreter (
      language <- o.language,
      className <- o.className
    )
}
abstract rule GenDomainModelNavigator2GenDomainModelNavigator {
  from
    o : Before!GenDomainModelNavigator
  to
    m : After!GenDomainModelNavigator (
      generateDomainModelNavigator <- o.generateDomainModelNavigator,
      domainContentExtensionID <- o.domainContentExtensionID,
      domainContentExtensionName <- o.domainContentExtensionName,
      domainContentExtensionPriority <- o.domainContentExtensionPriority,
      domainContentProviderClassName <- o.domainContentProviderClassName,
      domainLabelProviderClassName <- o.domainLabelProviderClassName,
      domainModelElementTesterClassName <- o.
     domainModelElementTesterClassName,
      domainNavigatorItemClassName <- o.domainNavigatorItemClassName
    )
}
rule GenNavigator2GenNavigator extends
     GenDomainModelNavigator2GenDomainModelNavigator {
  from
    o : Before!GenNavigator
  to
    m : After!GenNavigator (
      contentExtensionID <- o.contentExtensionID,
      contentExtensionName <- o.contentExtensionName,
      contentExtensionPriority <- o.contentExtensionPriority,
      linkHelperExtensionID <- o.linkHelperExtensionID,
      sorterExtensionID <- o.sorterExtensionID,
      actionProviderID <- o.actionProviderID,
      contentProviderClassName <- o.contentProviderClassName,
      labelProviderClassName <- o.labelProviderClassName,
      linkHelperClassName <- o.linkHelperClassName,
      sorterClassName <- o.sorterClassName,
      actionProviderClassName <- o.actionProviderClassName,
      abstractNavigatorItemClassName <- o.abstractNavigatorItemClassName,
      navigatorGroupClassName <- o.navigatorGroupClassName,
      navigatorItemClassName <- o.navigatorItemClassName,
      uriInputTesterClassName <- o.uriInputTesterClassName,
      packageName <- o.packageName,
      childReferences <- o.childReferences
    )
```

```
}
rule GenNavigatorChildReference2GenNavigatorChildReference {
  from
    o : Before!GenNavigatorChildReference
  to
    m : After!GenNavigatorChildReference (
      parent <- o.parent,
      child <- o.child,
      referenceType <- o.referenceType,
      groupName <- o.groupName,
      groupIcon <- o.groupIcon,
      hideIfEmpty <- o.hideIfEmpty
    )
}
rule GenNavigatorPath2GenNavigatorPath {
  from
    o : Before!GenNavigatorPath
  to
    m : After!GenNavigatorPath (
      segments <- o.segments
    )
}
rule GenNavigatorPathSegment2GenNavigatorPathSegment {
  from
    o : Before!GenNavigatorPathSegment
  to
    m : After!GenNavigatorPathSegment (
      from <- o.from,
      to <- o.to
    )
}
rule GenPropertySheet2GenPropertySheet {
  from
    o : Before!GenPropertySheet
  to
    m : After!GenPropertySheet (
      tabs <- o.tabs,
      packageName <- o.packageName,
      readOnly <- o.readOnly,
      needsCaption <- o.needsCaption,
      labelProviderClassName <- o.labelProviderClassName
    )
}
abstract rule GenPropertyTab2GenPropertyTab {
  from
    o : Before!GenPropertyTab
```

```
 to
  m : After!GenPropertyTab (
   iD <- o.iD,
   label <- o.label
  )
}
rule GenStandardPropertyTab2GenStandardPropertyTab extends
    GenPropertyTab2GenPropertyTab {
 from
  o : Before!GenStandardPropertyTab
 to
  m : After!GenStandardPropertyTab
}
rule GenCustomPropertyTab2GenCustomPropertyTab extends
    GenPropertyTab2GenPropertyTab {
 from
  o : Before!GenCustomPropertyTab
 to
  m : After!GenCustomPropertyTab (
   className <- o.className,
   filter <- o.filter
  )
}
abstract rule GenPropertyTabFilter2GenPropertyTabFilter {
 from
  o : Before!GenPropertyTabFilter
 to
  m : After!GenPropertyTabFilter
}
rule TypeTabFilter2TypeTabFilter extends
    GenPropertyTabFilter2GenPropertyTabFilter {
 from
  o : Before!TypeTabFilter
 to
  m : After!TypeTabFilter (
   types <- o.types,
   generatedTypes <- o.generatedTypes
  )
}
rule CustomTabFilter2CustomTabFilter extends
    GenPropertyTabFilter2GenPropertyTabFilter {
 from
  o : Before!CustomTabFilter
 to
  m : After!CustomTabFilter (
   className <- o.className
  )
```

```
}
abstract rule GenContributionItem2GenContributionItem {
  from
    o : Before!GenContributionItem
  to
    m : After!GenContributionItem
}
rule GenSharedContributionItem2GenSharedContributionItem extends
    GenContributionItem2GenContributionItem {
  from
    o : Before!GenSharedContributionItem
  to
    m : After!GenSharedContributionItem (
      actualItem <- o.actualItem
    )
}
rule GenGroupMarker2GenGroupMarker extends
    GenContributionItem2GenContributionItem {
  from
    o : Before!GenGroupMarker
  to
    m : After!GenGroupMarker (
      groupName <- o.groupName
    )
}
rule GenSeparator2GenSeparator extends
    GenContributionItem2GenContributionItem {
  from
    o : Before!GenSeparator
  to
    m : After!GenSeparator (
      groupName <- o.groupName
    )
}
rule GenActionFactoryContributionItem2GenActionFactoryContributionItem
    extends GenContributionItem2GenContributionItem {
  from
    o : Before!GenActionFactoryContributionItem
  to
    m : After!GenActionFactoryContributionItem (
      name <- o.name
    )
}
abstract rule GenContributionManager2GenContributionManager extends
    GenContributionItem2GenContributionItem {
  from
    o : Before!GenContributionManager
```

```
  to
    m : After!GenContributionManager (
      iD <- o.iD,
      items <- o.items
    )
}
rule GenMenuManager2GenMenuManager extends
    GenContributionManager2GenContributionManager {
  from
    o : Before!GenMenuManager
  to
    m : After!GenMenuManager (
      name <- o.name
    )
}
rule GenToolBarManager2GenToolBarManager extends
    GenContributionManager2GenContributionManager {
  from
    o : Before!GenToolBarManager
  to
    m : After!GenToolBarManager
}
rule GenApplication2GenApplication {
  from
    o : Before!GenApplication
  to
    m : After!GenApplication (
      iD <- o.iD,
      title <- o.title,
      packageName <- o.packageName,
      className <- o.className,
      perspectiveId <- o.perspectiveId,
      supportFiles <- o.supportFiles,
      sharedContributionItems <- o.sharedContributionItems,
      mainMenu <- o.mainMenu,
      mainToolBar <- o.mainToolBar
    )
}
```

Listing C.13: GMF Generator model migration in ATL

```
for (genLinkLabel in gen.GenLinkLabel.allInstances) {
  genLinkLabel.unset(notationViewFactoryClassName)
}

for (genLink in gen.GenLink.allInstances) {
  genLink.unset(notationViewFactoryClassName)
```

```
}

for (genEditorGenerator in gen.GenEditorGenerator.allInstances) {
  def genContextMenu = gen.GenContextMenu.newInstance()
  genEditorGenerator.contextMenus.add(genContextMenu)

  genContextMenu.context.add(genEditorGenerator.diagram)
  genContextMenu.items.add(gen.LoadResourceAction.newInstance())

  for (shortcutName in genContextMenu.diagram.containsShortcutsTo) {
    genContextMenu.items.add(gen.CreateShorcutAction.newInstance())
  }
}

for (genDiagram in gen.GenDiagram) {
  genDiagram.validationProviderPriority = gen.ProviderPriority#Lowest
}

for (featureLabelModelFacet in gen.FeatureLabelModelFacet) {
  def viewMethod = featureLabelModelFacet.unset(viewMethod)
  def editMethod = featureLabelModelFacet.unset(editMethod)
  featureLabelModelFacet.parser = createOrRetrievePredefinedParser(
     viewMethod, editMethod)
}

for (designLabelModelFacet in gen.DesignLabelModelFacet) {
  designLabelModelFacet.parser = createOrRetrieveExternalParser()
}


createOrRetrievePredefinedParser = { viewMethod, editMethod ->
  if (getPredefinedParser(viewMethod, editMethod) == null) {
    createOrRetrieveGenParsers().implementations.add(
     createPredefinedParser(viewMethod, editMethod))
  }

  return getPredefinedParser(viewMethod, editMethod)
}

getPredefinedParser = { viewMethod, editMethod ->
  return gen.PredefinedParser.allInstances.find{ it -> it.viewMethod ==
       viewMethod &amp;&amp; p.editMethod == editMethod }
}

createPredefinedParser = { viewMethod, editMethod ->
  def parser = gen.PredefinedParser.newInstance()
```

```
  parser.viewMethod = viewMethod
  parser.editMethod = editMethod
  return parser
}

createOrRetrieveExternalParser = {
  if (gen.ExternalParser.allInstances.size == 0) {
    createOrRetrieveGenParsers().implementations.add(gen.ExternalParser.newInstance())

  }

  return gen.ExternalParser.first
}

createOrRetrieveGenParsers = {
  if (gen.GenEditorGenerator.allInstances.first.labelParsers == null) {
    gen.GenEditorGenerator.allInstances.first.labelParsers = gen.GenParsers.newInstance()
    gen.GenEditorGenerator.allInstances.first.labelParsers.extensibleViaService = true
  }

  return gen.GenEditorGenerator.allInstances.first.labelParsers
}
```

Listing C.14: GMF Generator model migration in COPE

```
migrate GenLinkLabel {
  migrated.notationViewFactoryClassName := null;
}

migrate GenLink {
  migrated.notationViewFactoryClassName := null;
}

migrate GenEditorGenerator {
  migrated.contextMenus.add(new Migrated!GenContextMenu);
  migrated.contextMenus.first.context.add(migrated.diagram);

  migrated.contextMenus.first.items.add(new Migrated!LoadResourceAction);

  for (shortcutName in original.diagram.containsShortcutsTo) {
    migrated.contextMenus.first.items.add(new Migrated!CreateShortcutAction);
  }
```

```
}


migrate GenDiagram {
  migrated.validationProviderPriority := Migrated!ProviderPriority#Lowest;
}

migrate FeatureLabelModelFacet {
  migrated.parser := createOrRetrievePredefinedParser(migrated.viewMethod, migrated.editMethod);
  migrated.viewMethod := null;
  migrated.editMethod := null;
}

migrate DesignLabelModelFacet {
  migrated.parser := createOrRetrieveExternalParser();
}

operation createOrRetrievePredefinedParser(viewMethod : Any, editMethod : Any) : Migrated!PredefinedParser {
  if (getPredefinedParser(viewMethod, editMethod).isUndefined()) {
    createOrRetrieveGenParsers().implementations.add(createPredefinedParser(viewMethod, editMethod));
  }

  return getPredefinedParser(viewMethod, editMethod);
}

operation getPredefinedParser(viewMethod : Any, editMethod : Any) : Migrated!PredefinedParser {
  return Migrated!PredefinedParser.all.selectOne(p | p.viewMethod = viewMethod and p.editMethod = editMethod);
}

operation createPredefinedParser(viewMethod : Any, editMethod : Any) : Migrated!PredefinedParser {
  var parser := new Migrated!PredefinedParser;
  parser.viewMethod := viewMethod;
  parser.editMethod := editMethod;
  return parser;
}

operation createOrRetrieveExternalParser() : Migrated!ExternalParser {
  if (Migrated!ExternalParser.all.isEmpty()) {
    createOrRetrieveGenParsers().implementations.add(new Migrated!
```

```
    ExternalParser);
  }

  return Migrated!ExternalParser.all.first;
}

operation createOrRetrieveGenParsers() : Migrated!GenParsers {
  if (Migrated!GenEditorGenerator.all.first.labelParsers.isUndefined())
      {
    Migrated!GenEditorGenerator.all.first.labelParsers := new Migrated!
        GenParsers;
    Migrated!GenEditorGenerator.all.first.labelParsers.
      extensibleViaService := true;
  }

  return Migrated!GenEditorGenerator.all.first.labelParsers;
}
```

Listing C.15: GMF Generator model migration in Flock

Bibliography

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

[Ackoff 1962] Russell L. Ackoff. *Scientific Method: Optimizing Applied Research Decisions*. John Wiley and Sons, 1962.

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

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

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

[Apostel 1960] Leo Apostel. Towards the formal study of models in the non-formal sciences. *Synthese*, 12:125–161, 1960.

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

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

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

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

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

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

[Bézivin & Gerbé 2001] Jean Bézivin and Olivier Gerbé. Towards a precise definition of the OMG/MDA framework. In *Proc. ASE*, pages 273–280. IEEE Computer Society, 2001.

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

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

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

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

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

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

[Brooks 1986] Frederick P. Brooks. No silver bullet – essence and accident in software engineering (invited paper). In *Proc. International Federation for Information Processing*, pages 1069–1076, 1986.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[Dig 2007] Daniel Dig. *Automated Upgrading of Component-Based Applications*. PhD thesis, University of Illinois at Urbana-Champaign, USA, 2007.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[Feathers 2004] Michael C. Feathers. *Working Effectively with Legacy Code*. Prentice Hall, 2004.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[Halstead 1977] Maurice H. Halstead. *Elements of Software Science.* Elsevier Science Inc., 1977.

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

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

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

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

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

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

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

[INRIA 2007] INRIA. AMMA project page [online]. [Accessed 30 June 2008] Available at: `http://wiki.eclipse.org/AMMA`, 2007.

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

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

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

[Jackson 1995] M. Jackson. *Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices.* ACM Press, 1995.

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

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

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

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

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

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

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

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

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

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

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

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

[Kolovos *et al.* 2007b] Dimitrios S. Kolovos, Richard F. Paige, Louis M. Rose, and Fiona A.C. Polack. Bridging the Epsilon Wizard Language and the Eclipse Graphical Modeling Framework. In *Proc. Eclipse Summit*, Ludwigsburg, Germany, 2007.

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

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

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

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

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

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

[Kurtev 2004] Ivan Kurtev. *Adaptability of Model Transformations.* PhD thesis, University of Twente, Netherlands, 2004.

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

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

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

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

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

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

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

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

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

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

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

[Melnik 2004] Sergey Melnik. *Generic Model Management: Concepts and Algorithms.* PhD thesis, University of Leipzig, Germany, 2004.

[Mens & Demeyer 2007] Tom Mens and Serge Demeyer. *Software Evolution.* Springer-Verlag, 2007.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[openArchitectureWare 2008] openArchitectureWare. XPand Language Reference [online]. [Accessed 18 August 2010] Available at: `http://wiki.eclipse.org/AMMA`, 2008.

[Paige *et al.* 2007] Richard F. Paige, Phillip J. Brooke, and Jonathan S. Ostroff. Metamodel-based model conformance and multiview consistency checking. *ACM Transactions on Software Engineering and Methodology*, 16(3), 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.

[Patrascoiu & Rodgers 2004] Octavian Patrascoiu and Peter Rodgers. Embedding OCL expressions in YATL. In *Proc. OCL and Model-Driven Engineering Workshop*, 2004.

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

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

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

[Ramil & Lehman 2000] Juan F. Ramil and Meir M. Lehman. Cost estimation and evolvability monitoring for software evolution processes. In *Proc. Workshop on Empirical Studies of Software Maintenance*, 2000.

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

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

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

[Rose *et al.* 2009a] Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Enhanced automation for managing model and metamodel inconsistency. In *Proc. ASE*, pages 545–549. ACM Press, 2009.

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

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

[Rose *et al.* 2010b] Louis M. Rose, Dimitrios S. Kolovos, Nicholas Drivalos, James. R. Williams, Richard F. Paige, Fiona A.C. Polack, and Kiran J. Fernandes. Concordance: An efficient framework for managing model integrity [submitted to]. In *Proc. European Conference on Modelling Foundations and Applications*, volume 6138 of *LNCS*, pages 62–73. Springer, 2010.

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

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

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

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

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

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

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

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

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

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

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

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

[Starfield *et al.* 1990] M. Starfield, K.A. Smith, and A.L. Bleloch. *How to model it: Problem Solving for the Computer Age.* McGraw-Hill, 1990.

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

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

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

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

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

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

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

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

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

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

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

[Welch & Barnes 2005] Peter H. Welch and Fred R. M. Barnes. Communicating mobile processes. In *Proc. Symposium on the Occasion of 25 Years of Communicating Sequential Processes (CSP)*, volume 3525 of *LNCS*, pages 175–210. Springer, 2005.

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