COPE: Coupled Evolution of Metamodels and Models for the Eclipse Modeling Framework

Markus Herrmannsdoerfer[1], Sebastian Benz[2], and Elmar Juergens[1]

[1] Institut für Informatik
Technische Universität München
Boltzmannstr. 3, 85748 Garching b. München, Germany
{herrmama, juergens}@in.tum.de
[2] BMW Car IT GmbH
Petuelring 116, 80809 München, Germany
sebastian.benz@bmw-carit.de

Abstract. In consequence of changing requirements and technological progress, metamodels are subject to change. Manually migrating models to a new version of their corresponding metamodel is costly, tedious and error-prone. The coupled evolution of a metamodel and its models is a sequence of metamodel changes and their corresponding model migrations. These coupled changes are either metamodel-specific or metamodel-independent. Metamodel-independent changes can be reused to evolve different metamodels and their models, thus reducing migration effort. However, tool support is necessary in order to exploit these reuse opportunities. We propose a language based on the Eclipse Modeling Framework that allows for decomposition of a migration into manageable, modular coupled changes. It provides a reuse mechanism for metamodel-independent changes, but is at the same time expressive enough to cater for complex, metamodel-specific changes.

1 Introduction

Due to their high level of abstraction, modeling languages are a promising approach to decrease software development costs by increasing productivity. Significant work in both research and practice has been invested into tool support for the initial development of modeling languages. A very prominent example is the Eclipse Modeling Framework (EMF)[3]. As modeling languages are receiving increased attention in industry, their maintenance is gaining importance. Although often neglected, a language is subject to change like any other software artifact [1]. EMF and the Graphical Modeling Framework (GMF)[4] for instance, although relatively young, have already been adapted to technological progress or evolving requirements. A modeling language is evolved by adapting its metamodel to the new requirements. Existing models have to be migrated so that they can be used with the evolved modeling language.

[3] http://www.eclipse.org/modeling/emf/
[4] http://www.eclipse.org/modeling/gmf/

To better understand the nature of coupled evolution of metamodels and models in practice, we performed a study of the evolution history of two real-world metamodels [2]. The study's main goal was to determine to which degree coupled evolution can be automated in practice. Our results showed that there is substantial potential for reuse of coupled change operations, since more than three quarters of all coupled changes were not metamodel-specific and the corresponding model migration might be reused across different metamodels. A suitable library of coupled evolution operations hence can provide significant reduction of evolution effort. On the other hand, the remaining quarter of the coupled changes were metamodel-specific and therefore required a custom model migration. The analysis thus indicated that, in order to best support the sequence of metamodel-specific and -independent changes that make up language evolution, suitable tool support must satisfy two central requirements: *Reuse* of coupled evolution operations is required to take advantage of the high amount of recurring metamodel-independent changes. *Expressiveness* is required to cater for complex transformations involved in metamodel-specific coupled evolution operations.

Currently, to our best knowledge, there is no approach that combines both the desired level of expressiveness and reuse. To alleviate this, we present COPE, a language for the coupled evolution of metamodels and models that provides both reuse of recurring coupled evolution operations and the expressiveness to describe custom evolution steps. COPE offers an expressive language to specify the adaptation of the metamodel together with its corresponding model migration as coupled transactions. Generalization of coupled transactions allows for reuse of recurring coupled changes across metamodels. Coupled transactions are composeable in the sense that the evolution from one metamodel version to the next can be composed of manageable, modular transactions, thus allowing for flexible combination of reusable and custom individual coupled changes.

Outline. In Section 2, we introduce the concepts that form the basis of our coupled evolution language. In Section 3, we explain in more detail how the language concepts are implemented for use with the Eclipse Modeling Framework. In Section 4, we present the tools that were developed to support the application of COPE, before we conclude in Section 5.

2 Coupled Evolution of Metamodels and Models

A metamodel is changed in order to *adapt* to technological progress or changed requirements. Hence, existing models must be *migrated* in order to conform to the adapted metamodel. We refer to the combination of *metamodel adaptation* and *model migration* as *coupled evolution*. Figure 1 depicts the concept of coupled evolution, where each metamodel adaptation has a specific model migration.

The metamodel adaptation is usually performed manually in the modeling tool that is used to edit the metamodel. In contrast, model migration is encoded as a model transformation that transforms a model such that the new model

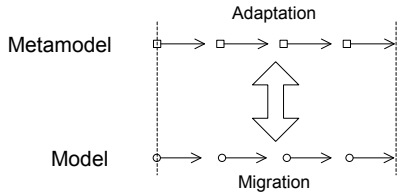

Fig. 1. Coupled evolution of metamodel and model

conforms to the evolved metamodel. There are multiple languages for model transformation that can be used to encode model migrations. In general, we distinguish between *exogenous* and *endogenous* model transformation, depending on whether source and target metamodel of the transformation are different or not [3]. Languages for exogenous model transformation usually require to specify the mapping of all elements from the source to the target metamodel. As typically only a subset of all metamodel elements are modified during a language evolution step, an exogenous transformation for model migration contains a high fraction of identity rules.

2.1 Coupled Transactions

Model migration is thus best served by a language that allows to combine the properties of both languages for exogenous and endogenous model transformation: one needs to be able to specify the transformation from a source metamodel to a different target metamodel, but only for the metamodel elements for which a migration is required. In order to achieve this, we propose to soften the conformance between a metamodel and its model during coupled evolution: the metamodel can first be adapted regardless of its models, and the model can then be migrated to the evolved metamodel. Therefore, COPE provides a number of expressive primitives to encode both metamodel adaptation and model migration independently of each other. However, softening the conformance during model migration comes at the price that a model may not always conform to its metamodel. In order to enforce conformance after a certain change to metamodel and model, we introduce the following notion: A *coupled transaction* is defined as an operation that evolves the metamodel and migrates the model such that the following properties hold:

Consistency preservation: The evolved metamodel is consistent, i. e. fulfills the constraints defined by the meta-metamodel, if the original one was.

Conformance preservation: The migrated model conforms to the evolved metamodel, if the original model conformed to the original metamodel.

Note that both consistency and conformance thus have to hold only at transaction boundaries, i. e. the metamodel may be inconsistent or the model may not conform to the metamodel during a transaction. Coupled transactions are

composeable by simply sequencing them. A comprehensive migration from one metamodel version to the next can thus be composed of a number of manageable coupled transactions. Each coupled transaction is modular, i. e. can be specified independently of any other coupled transaction.

2.2 Expressiveness and Reuse through Coupled Transactions

Our study of the evolution of industrial metamodels [2] showed that the evolution of a language can, in principle, be split into individual coupled changes, each denoting a specific metamodel adaptation and the corresponding model migration. Coupled transactions offer a formalism to specify such individual coupled evolution operations. Furthermore, coupled transactions offer an apt way of satisfying the central requirements identified in [2] for efficient tool support for coupled evolution of metamodels and models:

Expressiveness: In order to cater for arbitrarily complex model migrations, specification formalisms must be sufficiently expressive.

Reuse: A substantial amount of changes were not metamodel-specific and occurred during the evolution of different metamodels. In order to avoid repeated specification of recurring changes, a reuse mechanism for coupled changes is necessary.

In order to fulfill the stated requirements, we provide two kinds of coupled transactions: reusable and custom coupled transactions. A *reusable coupled transaction* allows to reuse recurring coupled evolution operations across metamodels and has thus to be specified independently of a specific metamodel. We can define a library of reusable coupled transactions which can be invoked by a language developer, thus promising to significantly reduce effort associated with metamodel adaptation and migration encoding. However, not every coupled evolution can be formulated only with reusable coupled transactions available in a library. For this reason, a *custom coupled transaction* can be manually defined by the language developer for complex migrations that are specific to a metamodel. Through the combination of arbitrarily expressive custom coupled transactions and reusable coupled transactions, as is depicted in Figure 2, composeability enables us to combine both expressiveness and reuse.

3 COPE

In this section, we present COPE, our language for the coupled evolution of metamodels and models based on the Eclipse Modeling Framework. In order to achieve in-place transformation, COPE softens the conformance of a model to its corresponding metamodel during coupled evolution. Based on this decoupling of metamodel and model, COPE provides expressive primitives for both metamodel adaptation and model migration. These primitives can be combined to encode custom and reusable coupled transactions.

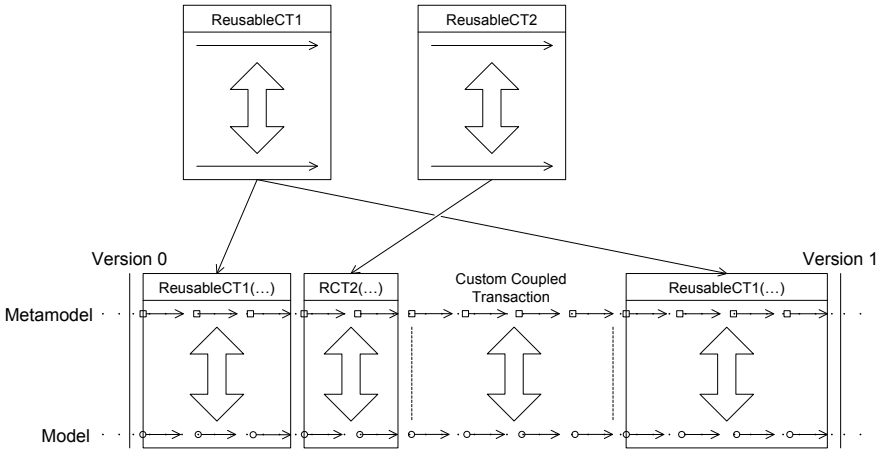

Fig. 2. Composability of coupled transactions

3.1 Decoupling Metamodel and Model

Figure 3 depicts the relationship between a model and its metamodel. Inside transaction boundaries, model and metamodel can be modified independently of each other, whereas conformance is required at transaction boundaries. As a consequence, we are able to perform in-place transformation, i. e. direct updates of the models. In-place transformation is more efficient than out-of-place transformation, which requires to rebuild the migrated model from scratch.

A metamodel is defined by means of Ecore, the meta-metamodel of EMF. The metamodel is consistent if it fulfills the constraints defined by Ecore. A model consists of a number of instances (Instance). Each instance has a number of slots (Slot) which are the valuations of either attributes (AttributeSlot) or association ends (ReferenceSlot). Instances and slots are associated to their corresponding metamodel elements. However, these associations do not constrain an instance to always conform to its type in the metamodel. This loose association allows us to first modify the metamodel without affecting the model and then migrating the model to the evolved metamodel. Since this decoupling can lead to states where the model does not conform to its metamodel, conformance is checked at transaction boundaries by means of a validator.

3.2 Primitives for Metamodel Adaptation and Model Migration

COPE provides both expressive primitives to specify metamodel adaptation and to specify model migration.

Metamodel adaptation. For metamodel adaptation, COPE provides the following primitives to query the metamodel:

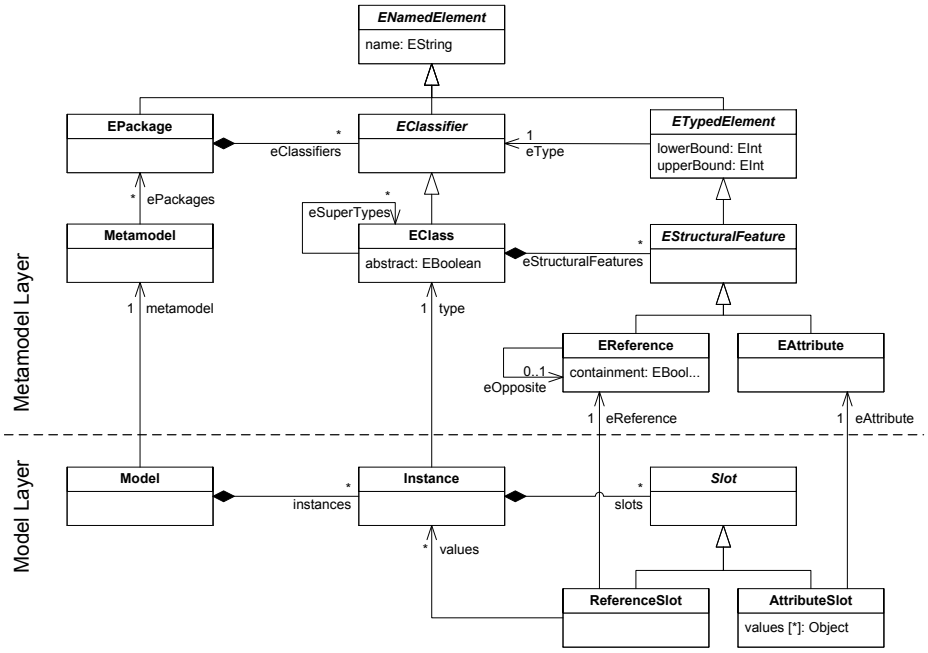

Fig. 3. Association between metamodel and model

- `<qualifiedName>` to access a metamodel element by means of its qualified name, i.e. a package, class or feature.
- `<element>.<featureName>` to access the value of a feature of a metamodel element as defined by the meta-metamodel.

COPE provides the following primitives to modify the metamodel in-place:

- `<package>.newClass(...)`, `<class>.newAttribute(...)` or `<class>.newReference(...)` to create a new class, attribute or reference.
- `<element>.delete()` to delete a metamodel element.
- `<element>.<featureName> = <value>` to modify the value of a feature of a metamodel element.

Model migration. For model migration, COPE provides the following primitives to query a model:

- `<class>.instances` to access all instances of a class.
- `<class>.allInstances` to access all instances of a class or any of its subclasses.
- `<instance>.get(<feature>)` or `<instance>.<featureName>` to access the value of a feature of an instance (the short form can be used if the feature with that name is available in the instance's type).

COPE provides the following primitives to modify the model in-place:

- `<class>.newInstance()` to create a new instance of a class.
- `<instance>.delete()` to delete an instance from the model.
- `<instance>.migrate(<class>)` to change the type of an instance to a different class.
- `<instance>.set(<feature>, <value>)` or `<instance>.<featureName > = <value>` to modify the value of a feature of an instance.
- `<instance>.unset(<feature>)` to unset and return the value of a feature of an instance.

These primitives are constructed in a way that they also allow to access model information which currently does not conform to the metamodel.

3.3 Coupled Transactions

The primitives can be invoked from within the general-purpose scripting language Groovy[5] in order to take advantage of its expressiveness. The interpreter of COPE ensures that a coupled transaction can only be successfully completed in case it preserves consistency and conformance.

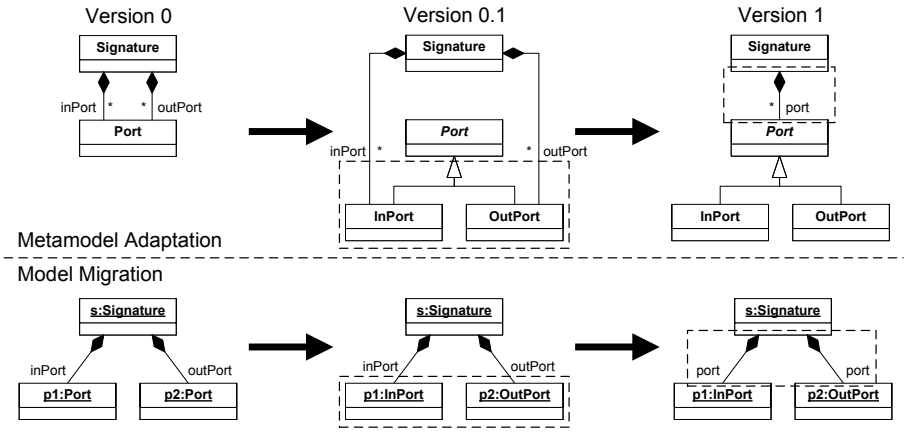

Fig. 4. Coupled evolution of example metamodel and model

Custom coupled transaction. Custom coupled transactions are coupled transactions that are specific to a certain metamodel. A custom coupled transaction is specified by the language developer as a script that uses a number of primitives to specify both metamodel adaptation and model migration. Figure 4 depicts a simple example metamodel and the two coupled changes we plan to perform. The metamodel allows to express Signatures of components which consist of input

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

and output Ports (references inPort and outPort). In version 0 of the metamodel, a port does not by itself know whether it is an input or output port. In order to introduce the missing information in version 0.1, we refine the class Port into specialized subclasses InPort and OutPort and make it abstract. Instances of Port have to be migrated depending on whether they are input and output ports of the signature. As there is not yet a reusable coupled transaction for this coupled evolution, we have to manually encode both metamodel adaptation and model migration in a custom coupled transaction that is shown in Listing 1.

Listing 1. Custom coupled transaction

```
// metamodel adaptation
Signature.inPort.eType = newClass("InPort", [Port])
Signature.outPort.eType = newClass("OutPort", [Port])
Port.'abstract' = true
// model migration
for(signature in Signature.instances) {
  for(port in signature.inPort) port.migrate(InPort)
  for(port in signature.outPort) port.migrate(OutPort)
}
```

Reusable coupled transaction. We use Groovy's reuse mechanism of procedures in order to declare reusable coupled transactions. Reusable coupled transactions can be instantiated by simply invoking the corresponding procedure. The applicability of a reusable coupled transaction can be restricted by constraints in the form of assertions. Since we have introduced specialized classes for input and output ports, we no longer need to distinguish them through the references from Signature in version 1 of the metamodel (see Figure 4). We can now merge the two references into a single reference which is performed by means of an existing reusable coupled transaction from a library. The declaration of the reusable coupled transaction that merges one reference into another, is depicted in Listing 2. Listing 3 shows how to instantiate the reusable coupled transaction in order to merge the references inPort and outPort into the reference port created before.

4 Tool Support

In order to ease the application of COPE, we have integrated it to the metamodel editor provided by EMF. A screenshot of the extended metamodel editor is shown in Figure 5.[6] The user interface provides facilities to perform the coupled evolution of metamodel and model. A reusable coupled transaction can

[6] A screencast demonstrating the capabilities of the tool is available at `http://wwwbroy.in.tum.de/~herrmama/cope/pmwiki.php?n=Demo.Main`.

Listing 2. Declaration of a reusable coupled transaction

```
merge = {EReference toMerge, EReference mergeTo ->
  def contextClass = toMerge.eContainingClass
  // constraints
  assert contextClass.eAllStructuralFeatures.contains(mergeTo)
  assert toMerge.many && mergeTo.many
  assert toMerge.eReferenceType == mergeTo.eReferenceType ||
    toMerge.eReferenceType.eAllSuperTypes.contains(toMerge.
      eReferenceType)
  // metamodel adaptation
  toMerge.delete()
  // model migration
  for(instance in contextClass.allInstances) {
    instance.get(mergeTo).addAll(instance.unset(toMerge))
}
```

Listing 3. Instantiation of a reusable coupled transaction

```
Signature.newReference("port", Port, 0, -1, true)
merge(Signature.inPort, Signature.port)
merge(Signature.outPort, Signature.port)
```

be invoked through the *operation browser*. The browser is context-sensitive, i. e. offers only those reusable coupled transactions which are applicable to the elements currently selected in the metamodel editor. The browser allows to set the parameters of a reusable coupled transaction, and gives feedback on its applicability based on the constraints. When a reusable coupled transaction is executed, its invocation is tracked in an explicit *language history*. In case no reusable coupled transaction is available for the coupled evolution at hand, the language developer can perform a custom coupled transaction. First, the metamodel is directly authored in the EMF editor. The tool automatically tracks the metamodel changes in the history. A migration can later be attached to the sequence of metamodel changes by encoding it in the language presented in Section 3. The browser provides a release button to create a major version of the metamodel. After release, the language developer can initiate the automatic generation of a *migrator* that allows for the batch migration of models.

5 Conclusion

Just as other software artifacts, modeling languages evolve. To allow for efficient language evolution in practice, tool support must allow the specification of both expressive and reusable metamodel adaptation and model migration operations. In this paper, we have outlined a language that allows to compose a comprehensive migration from one metamodel version to the next of modular,

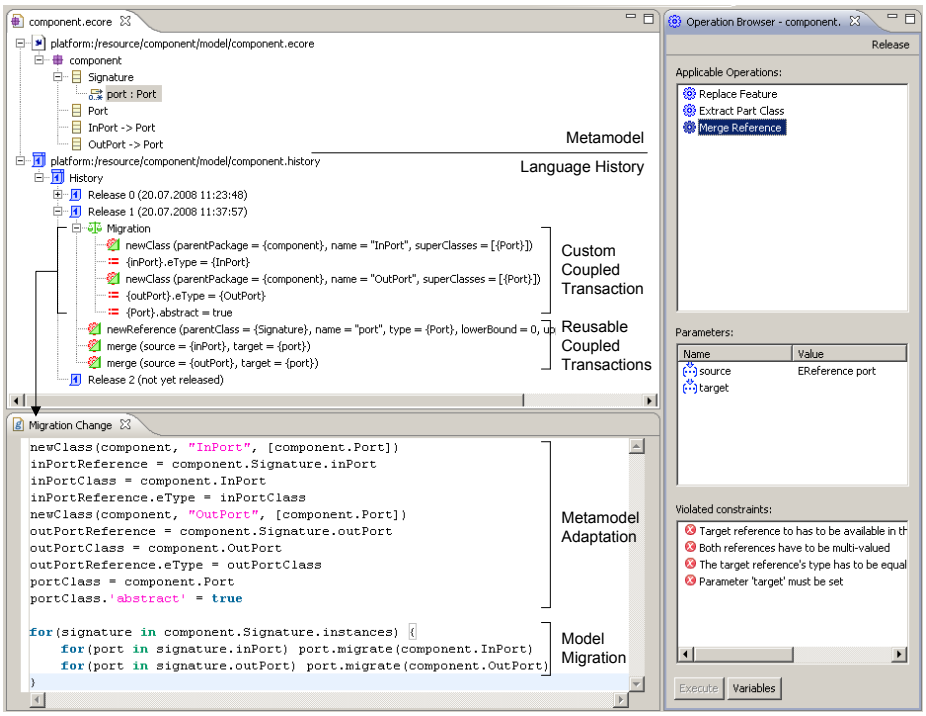

Fig. 5. Editor integration

coupled transactions. Reuse is provided by an abstraction mechanism that allows to encapsulate recurring migration knowledge. Composeability allows to easily combine reusable coupled transactions with custom coupled transactions, combining high migration development productivity and expressiveness. We implemented versioning for the coupled evolution of metamodel and models, and integrated the language implementation into the EMF metamodel editor. We are currently planing to contribute the language implementation, the versioning mechanism and the editor integration to the Eclipse Modeling Project.

References

1. Favre, J.M.: Languages evolve too! changing the software time scale. In: IWPSE. (2005)
2. Herrmannsdoerfer, M., Benz, S., Juergens, E.: Automatability of coupled evolution of metamodels and models in practice. In: MODELS. (2008)
3. Mens, T., Van Gorp, P.: A taxonomy of model transformation. (2005)