COPE: A Language for the Coupled Evolution of Metamodels and Models

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. Domain-specific modeling promises to increase productivity by offering modeling languages tailored to a problem domain. Such modeling languages are typically defined by a metamodel. In consequence of changing requirements and technological progress, the problem domains and thus the metamodels are subject to change. Manually migrating models to a new version of their corresponding metamodel is costly, tedious and error-prone and heavily hampers cost-efficient model-based development in practice. 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 which leads to reduction of migration effort. Tool support is necessary in order to benefit from potential reuse. We propose a language 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. Consequently, a variety of metamodel-based approaches for the development of modeling languages, such as Model-Driven Architecture [1], Software Factories [2] and Model-Integrated Computing [3] have been proposed in recent years. Significant work in both research and practice has been invested into tool support for the initial development of modeling languages. 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 [4]. This holds even for general-purpose languages: e. g. Java,

although relatively young, already has a rich evolution history. Domain-specific modeling languages are even more prone to change, as they have to be adapted whenever their domain changes due to technological progress or evolving requirements. A modeling language is evolved by adapting its metamodel to the new requirements. As other components like editors and interpreters depend on the metamodel, they have to be reconciled with the evolved metamodel. Furthermore, existing models have to be migrated so that they can be used with the evolved modeling language. Since the number of existing models of a successful modeling language typically outnumbers the number of editors or interpreters, model migration effort dwarfs tool reconciliation effort.

In order to circumvent model migration in current practice, language evolution is often performed in a downwards-compatible fashion. In other words, the language is changed in a way that the old models can still be used with the new language version without migration. However, downward compatibility heavily constrains language evolution and threatens language simplicity and quality, since preservation of all old language constructs can unnecessarily clutter and complicate a language. A prominent example is the introduction of Generics to Java, where backward compatibility is achieved by erasing all information about generic types during compilation. However, this technique leads to a lot of limitations and exceptions in applying generic types, as different types are treated uniformly during runtime by the Generics-unaware Java virtual machine [5]. To overcome restrictions of downward-compatible language evolution, better support for coupled evolution is required. Consequently, coupled evolution has been identified as one of the central challenges in software evolution [6].

To better understand the nature of coupled evolution[3] of metamodels and models in practice, we performed a study of the evolution history of two real-world metamodels [7]. It confirmed that metamodels do evolve in practice and that most metamodel changes require a migration of existing models. The study's main goal was to determine substantiated requirements for tool-support for coupled evolution. More specifically, we investigated whether reuse of coupled evolution operations can significantly reduce evolution effort. To this end, we categorized coupled changes into *metamodel-specific* and *-independent* changes. When a change is metamodel-specific, the corresponding model migration is as well, else, it can be reused across metamodels. Our results showed that there is large potential for reuse of coupled change operations, since more than three quarters of all coupled changes were not metamodel-specific. A suitable library of coupled evolution operations can thus 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

[3] Throughout the paper, we use the term *coupled evolution* instead of the term *co-evolution*, as we feel it better conveys the notion of coupling.

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 identify the need for a new language by analyzing existing approaches to automate model migration in response to metamodel adaptation, and motivate COPE by related work from schema evolution. We introduce the concepts that form the basis of our coupled evolution language in Section 3. In Section 4, we explain in more detail how the language concepts are implemented for use with the Eclipse Modeling Framework [8]. In Section 5, we present the tools that were developed to support the application of COPE. We conclude and state directions for future work in Section 6.

2 Related Work

Recently, the literature provides some work that tackles the problem of metamodel evolution. As a first work, Sprinkle discusses the problem of metamodel adaptation and highlights the conceptual differences between model migration and model transformation [9]: contrary to a transformation, a migration needs to be only specified for the metamodel elements which have actually changed. In order to reduce the effort for model migration, the author proposes a visual, graph-transformation based language for the specification of model migration [10]. While the language allows to compose a comprehensive migration of manageable steps, it does not provide a mechanism to reuse recurring migration knowledge. Gruschko et al. envision to automatically derive a model migration from the difference between two metamodel versions [11, 12]. The approach provides reuse by attaching a default migration for most primitive changes, but leaves open how complex migrations are handled. Wachsmuth adopts ideas from grammar engineering and proposes a classification of metamodel changes based on instance preservation properties [13]. Based on that, the author plans to provide a set of high-level primitives that are able to adapt the metamodel as well as to migrate models. While the approach allows to easily compose the high-level primitives, it does not provide an expressive language for the specification of complex migrations. In a nutshell, existing approaches to automate model migration either focus on expressiveness ***or*** reuse.

There are several areas in computer science that are subject to the problem of coupled transformation [14], e.g. schema evolution, grammar evolution and format evolution. As the problem of schema evolution has been a field of study for several decades, it has probably received the closest investigation. The history of the proposed approaches demonstrates a progression in terms of expressiveness and reuse which we want to outline here by means of representative examples. For the ORION database system, Banerjee et al. propose a fixed set of primitives that perform coupled evolution of the schema and data [15]. While highly reusable, their approach is limited to local schema restructuring. To allow for non-local changes, Ferrandina et al. propose separate languages for schema and instance data adaptation for the O_2 database system [16]. While more expressive, their approach does not allow for reuse of coupled transformation knowledge. In order to reuse recurring complex coupled evolutions, SERF, as proposed by Claypool et al., offers a mechanism to define arbitrary new high-level primitives [17], thus achieving both expressiveness ***and*** reuse. The combination of an expressive language to specify change and an abstraction mechanism to provide reuse, as proposed by SERF, inspired the language we propose in this paper.

3 Coupled Evolution of Metamodels and Models

A metamodel evolves, when it is modified in order to *adapt* to a changed problem domain. 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.

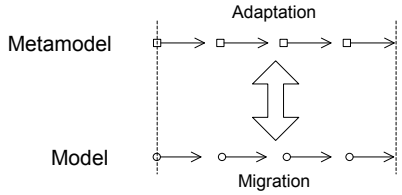

Fig. 1. Coupled evolution of metamodel and model

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 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, based on whether source and target metamodel of the transformation are different or

not [18]. 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. A language for endogenous model transformation is better suited to the nature of model migration, as it has to address only those metamodel elements for which the model needs to be modified. However, endogenous transformations require the source and the target metamodel to be the same which is not the case during language evolution.

3.1 Coupled Transactions

For this reason, model migration is best served by a language that allows to combine the properties of both languages for exogenous and endogenous model transformation: one need to be able to specify the transformation from a source metamodel to a different target metamodel, but only for the metamodel elements for which a migration is required. 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.

3.2 Expressiveness and Reuse through Coupled Transactions

Our study of the evolution of industrial metamodels [7] 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 [7] 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 decomposed of 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 which 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.

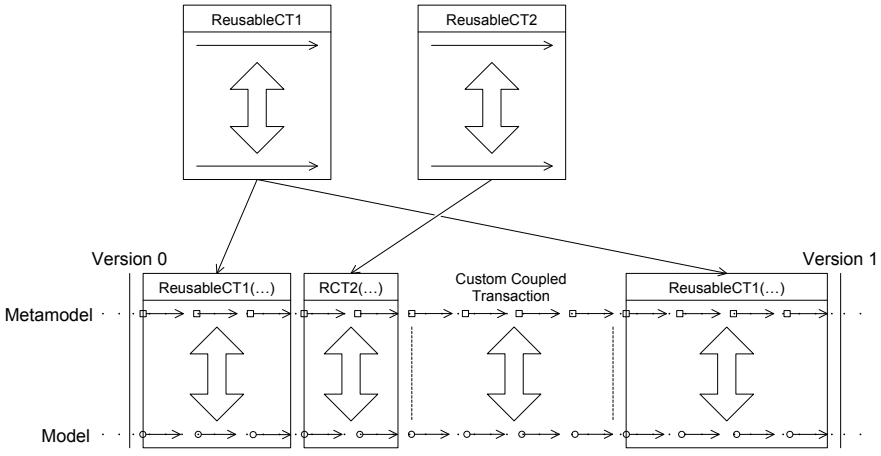

Fig. 2. Composability of coupled transactions

4 COPE

In this section, we present COPE, our language for the coupled evolution of metamodels and models. COPE implements the concept of coupled transactions and is based on the Eclipse Modeling Framework (EMF) [8]. 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 a number of expressive primitives to adapt the metamodel and to migrate the model independently of each other. These primitives can be combined to encode custom and reusable coupled transactions, which both require metamodel consistency and model conformance at their boundaries.

4.1 Decoupling Metamodel and Model

Figure 3 depicts the relationship between a model and its metamodel. Inside transactions 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 an in-place transformation of the model, i.e. the model is directly updated. In-place transformation is more efficient than out-of-place transformation, which requires to rebuild the migrated model from scratch.

As meta-metamodel, we use Ecore from the Eclipse Modeling Framework (EMF) [8]. However, our approach is not restricted to Ecore, as it can be transfered to all object-oriented metamodeling formalisms. A metamodel consists of a number of packages (EPackage) in whose scope classes are defined (EClass). A class may be abstract and may have a number of supertypes (eSuperTypes), thus supporting multiple inheritance. Each class defines a number of features (EStructuralFeature) which may be either attributes (EAttribute) or association ends (EReference), and which have a type and multiplicity (lowerBound, upperBound). Bidirectional associations are achieved by combining two association ends through eOpposite. Whole-part associations are modeled by so-called containment references.

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.

Consistency. The metamodel is consistent if it fulfills the constraints defined by the Ecore meta-metamodel. Examples for constraints are that no two classes may have matching names, or that a class may neither directly nor transitively

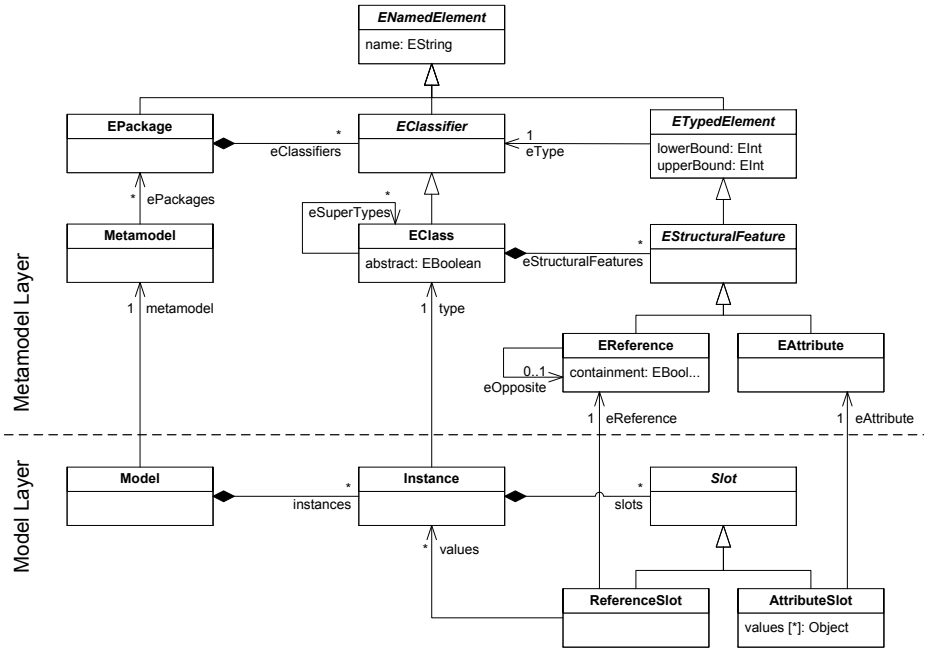

Fig. 3. Association between metamodel and model

be a supertype of itself. We refer the reader to the Java documentation of the EMF source code for a complete list of the constraints[4]. EMF provides a facility to easily check for the violation of these constraints which we employ in the implementation of COPE.

Conformance. The loose association between metamodel and model may lead to states where the model does not conform to its metamodel. However, we can define what it means for a model to conform to its metamodel based on the association between metamodel and model elements.

- A model conforms to its metamodel if each instance conforms to its type which has to be a non-abstract class defined by the model's metamodel.
- An instance conforms to its type if
  - each slot conforms to its feature which is defined by the instance's type or its super types,
  - for each mandatory[5] feature defined by the instance's type there is a corresponding slot,
  - either it is a root element of the model or it is referenced exactly once by a containment reference slot of another instance, and

[4] The source distribution of EMF can be downloaded from `http://www.eclipse.org/modeling/emf/`.

[5] A feature is mandatory if it is required to be set because of a non-zero lower bound.

  - it fulfills all further constraints which are defined in the context of the instance's type or its super types.
- An attribute slot conforms to its attribute if the value of the slot is consistent with the attribute's type and multiplicity.
- A reference slot conforms to its reference if
  - the value of the slot is consistent with the attribute's type and multiplicity, and
  - the instance belongs to the opposite reference slot of each value.

While on purpose not enforced by the loose association between metamodel and model, these constraints can be checked at transaction boundaries.

4.2 Primitives for Metamodel Adaptation and Model Migration

COPE provides a number of expressive primitives to specify metamodel adaptation and to specify model migration. The primitives are complete in the sense that every possible metamodel adaptation and every possible model migration can be encoded.

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

- `<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 that perform an in-place transformation:

- `<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>.exactInstances` to access all instances of a class.
- `<class>.instances` 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 that perform an in-place transformation:

- `<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 (the short form can be used if the feature with that name is available in the instance's type).
- `<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.

4.3 Coupled Transactions

The primitives can be invoked from within the general-purpose scripting language Groovy [19] in order to take advantage of its expressiveness. As a consequence, we can rely on the Turing-completeness of the host language to be able to describe complex coupled evolution operations. The interpreter of COPE ensures that a coupled transaction can only be successfully completed in case it preserves consistency and conformance.

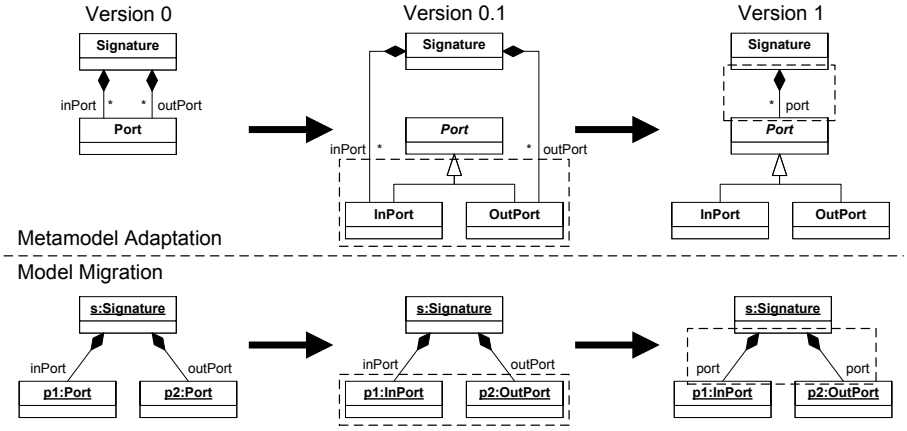

Fig. 4. Coupled evolution of example metamodel and model

Custom coupled transaction. Custom coupled transactions are coupled transactions which 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[6]. The metamodel allows to express Signatures of components which consist of input 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 according to the information 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 which 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 the reuse mechanism of procedures of the host language 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[7]. 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.

[6] For better overview, modified elements are highlighted by a dashed box.

[7] The token −> is part of the Groovy syntax and separates the parameter list from the body of a procedure.

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 reusable coupled transactions

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

5 Tool Support

On top of the language implementation, we developed further tool support to ease the application of COPE. Figure 5 provides an overview of the tool workflow using the example from Section 4.3. Reusable coupled transactions have to be declared independently of the specific metamodel, i. e. on the level of the metametamodel. Reusable coupled transactions can be registered to an explicit *library* through which they are made available to the language developer. The library is aware of the signature and the constraints of the reusable coupled transactions. All changes performed on the metamodel are maintained in an explicit *language history*. The history not only contains the metamodel adaptation, but also the encoded model migration. The history is structured according to the major language versions, i. e. when the language was deployed. All previous versions of the metamodel can be easily reconstructed from the information available in the history. In Figure 5, version 1 consists of the coupled transactions we performed in Section 4.3. A *migrator* can be generated from the language history that allows for the batch migration of models. The migrator can be invoked to migrate a model conforming to an earlier version of the metamodel to the newest version of the metamodel. This activity is fully automated, i.e. no information is required during migration.

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

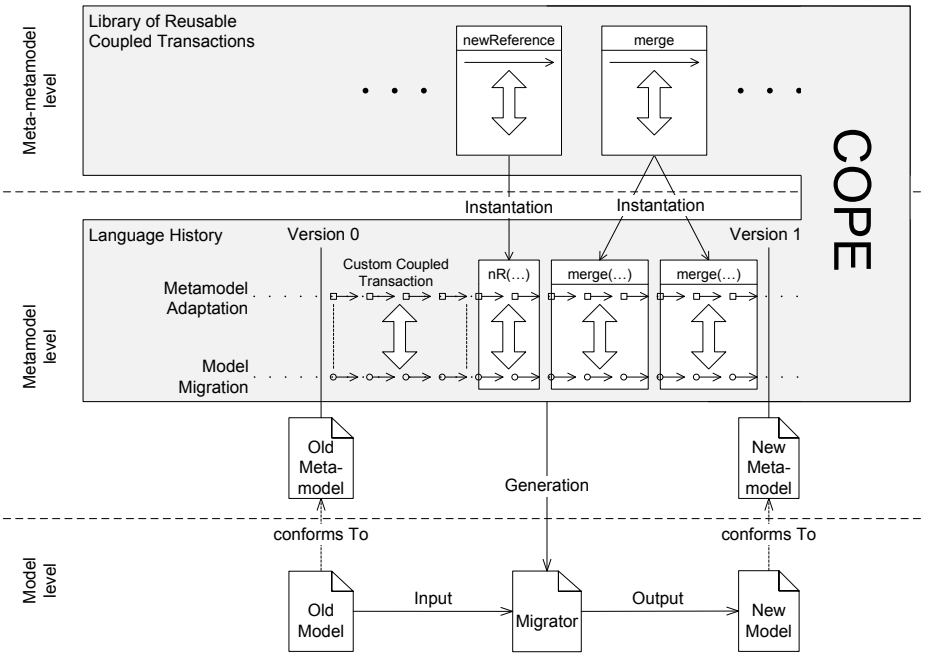

Fig. 5. Tool workflow

is shown in Figure 6[8]. The hands-on user interface provides facilities to perform the coupled evolution of metamodel and model. A reusable coupled transaction can 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 based on their type, and gives feedback on its applicability based on the constraints. When a reusable coupled transaction is executed, its invocation is tracked in the language history. Figure 6 shows the merge operation being available in the browser, and the executed merge operations stored in the 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 4. Figure 6 shows the model migration attached to the manual changes in a separate editor with syntax highlighting. 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.

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

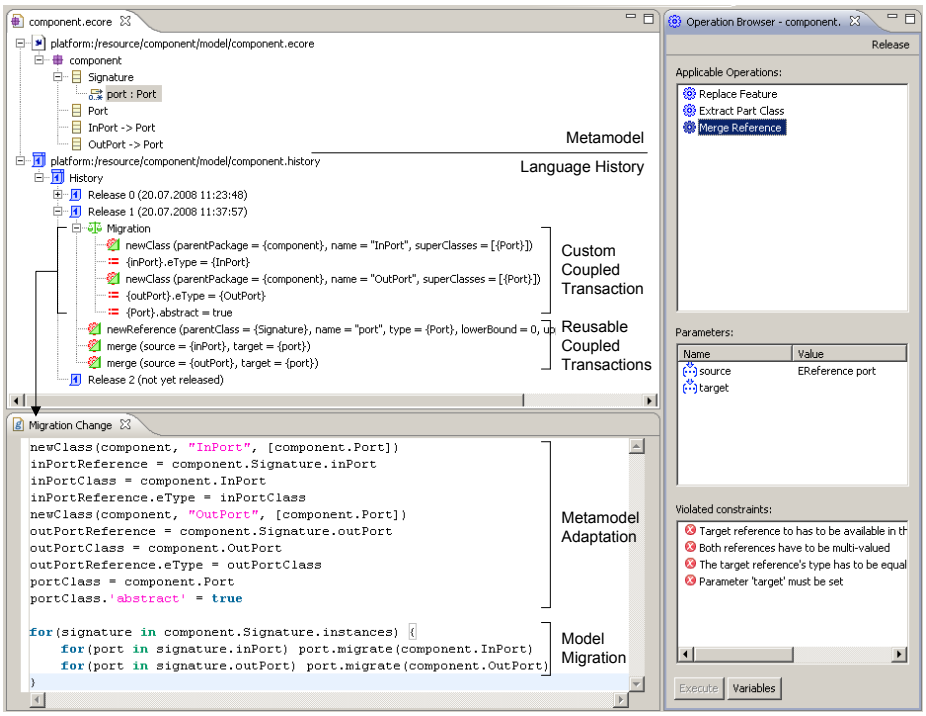

Fig. 6. Editor integration

6 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, coupled transactions. Expressiveness is provided by embedding the language into a Turing-complete host language. 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.

We intend to direct our future work on COPE in the following two directions: Based on our analysis of the coupled evolution history of two industrial metamodels [7], we plan to compile a library of reusable coupled transactions. We intend to apply and evaluate it in the context of a case study from the automo-

tive domain. Furthermore, we plan to develop a static analysis that allows us to verify conformance preservation of coupled transactions in a model-independent way, as the current version can only validate conformance preservation after transaction execution on a concrete model.

Moreover, we envision to employ our language for the iterative development of modeling languages. A version of a modeling language is created and deployed to be assessed by the modelers. The feedback of the modelers can then be easily incorporated into a new version of the modeling language which is again deployed for further assessment.

References

1. Kleppe, A., Warmer, J., Bast, W.: MDA Explained. The Model Driven Architecture: Practice and Promise. Addison-Wesley (2003)
2. Greenfield, J., Short, K., Cook, S., Kent, S., Crupi, J.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley (2004)
3. Sprinkle, J.: Model-integrated computing. IEEE Potentials (2004)
4. Favre, J.M.: Languages evolve too! changing the software time scale. In: IWPSE. (2005)
5. Allen, E., Cartwright, R.: The case for run-time types in generic java. In: PPPJ/IRE. (2002)
6. Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., Jazayeri, M.: Challenges in software evolution. In: IWPSE. (2005)
7. Herrmannsdoerfer, M., Benz, S., Juergens, E.: Automatability of coupled evolution of metamodels and models in practice. In: MODELS. (2008) (to appear) `http://wwwbroy.in.tum.de/~herrmama/cope/pmwiki.php?n=Publications.Main`.
8. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: Eclipse Modeling Framework. Addison-Wesley Professional (2003)
9. Sprinkle, J.M.: Metamodel driven model migration. PhD thesis (2003)
10. Sprinkle, J., Karsai, G.: A domain-specific visual language for domain model evolution. Journal of Visual Languages and Computing (2004)
11. Becker, S., Goldschmidt, T., Gruschko, B., Koziolek, H.: A process model and classification scheme for semi-automatic meta-model evolution. In: Workshop 'MDD, SOA and IT-Management'. (2007)
12. Gruschko, B., Kolovos, D., Paige, R.: Towards synchronizing models with evolving metamodels. In: CSMR. (2007)
13. Wachsmuth, G.: Metamodel adaptation and model co-adaptation. In: ECOOP. (2007)
14. Lämmel, R.: Coupled Software Transformations (Extended Abstract). In: First International Workshop on Software Evolution Transformations. (2004)
15. Banerjee, J., Kim, W., Kim, H.J., Korth, H.F.: Semantics and implementation of schema evolution in object-oriented databases. In: SIGMOD. (1987)
16. Ferrandina, F., Meyer, T., Zicari, R., Ferran, G., Madec, J.: Schema and database evolution in the O2 object database system. In: VLDB. (1995)
17. Claypool, K.T., Jin, J., Rundensteiner, E.A.: SERF: schema evolution through an extensible, re-usable and flexible framework. In: CIKM. (1998)
18. Mens, T., Van Gorp, P.: A taxonomy of model transformation. In: GraMoT. (2005)
19. Koenig, D., Glover, A., King, P., Laforge, G., Skeet, J.: Groovy in Action. Manning Publications Co. (2007)