COPE - Automating 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. Model-based development promises to increase productivity by offering modeling languages tailored to a specific domain. Such modeling languages are typically defined by a metamodel. In response to changing requirements and technological progress, the domains and thus the metamodels are subject to change. Manually migrating existing models to a new version of their metamodel is tedious and error-prone. Hence, adequate tool support is required to support the maintenance of modeling languages. This paper introduces COPE, an integrated approach to specify the coupled evolution of metamodels and models to reduce migration effort. With COPE, a language is evolved by incrementally composing modular coupled transformations that adapt the metamodel and specify the corresponding model migrations. This modular approach allows to combine the reuse of recurring transformations with the expressiveness to cater for complex transformations. We demonstrate the applicability of COPE in practice by modeling the coupled evolution of two existing modeling languages.

1 Introduction

Model-based development promises to increase productivity by offering modeling languages tailored to a specific domain. Consequently, a variety of metamodel-based approaches for the development of modeling languages, such as Model-Driven Architecture [1], Software Factories [2] and Domain-Specific Modeling [3] have been proposed in recent years. In response, modeling languages are receiving increased attention in industry. The AUTOSAR standard, for instance, defines a modeling language to specify automotive software architectures [4]. With the integration of modeling languages into industrial development practice, their maintenance is gaining importance. Although significant work in both academia and industry has been invested into tool support for the initial development of modeling languages, issues related to their maintenance are still largely disregarded.

Even though often neglected, a language is subject to change like any other software artifact [5]. This holds for both general-purpose and domain-specific modeling languages. For instance, UML [6] – a general purpose modeling language – already has a rich evolution history, although it is relatively young. Domain-specific modeling languages like e. g. AUTOSAR 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 evolved requirements. Due to metamodel adaptation, existing models may no longer conform to the adapted metamodel. These models have to be *migrated* so that they can be used with the evolved modeling language. Throughout the paper, the combination of metamodel adaptation and reconciling model migration is referred to as *coupled evolution*. Manually migrating existing models to the adapted metamodel is tedious and error-prone. Consequently, in current practice two approaches are used to handle evolution of modeling languages.

The first approach advocates to perform language evolution in a downwards-compatible fashion. In other words, the metamodel is adapted in a way that the old models can still be used with the evolved modeling language without migration. However, downward compatibility heavily constrains the way in which a metamodel can be adapted. Furthermore, the preservation of old constructs can unnecessarily clutter and complicate a metamodel. This approach can be further refined by using deprecation to signal metamodel changes. More precisely, constructs are marked deprecated, before they are actually removed from the metamodel. Users of the modeling language are then informed about the deprecated constructs which should no longer be used. However, deprecation shifts the responsibility for model migration from the developer of the modeling language to its users. In addition, deprecation also clutters and complicates the metamodel, as it leads to non-orthogonal constructs being available at the same time. In a nutshell, both downwards compatibility and deprecation heavily threaten the simplicity and quality of the metamodel. As a lot of artifacts like language editors and interpreters depend on the metamodel, these approaches also affect their simplicity and quality.

The second approach is to adapt the metamodel in a breaking fashion and to later implement a migrator, i. e. when the new version of the modeling language is deployed. The migrator preserves the information of an existing model by transforming it into a new version that conforms to the adapted metamodel. This approach has the advantage that the metamodel can be adapted in a clean manner, because legacy constructs can be removed. However, implementation of a migrator after a number of metamodel adaptations is tedious, as the developers of the modeling language have to become clear about the intentions behind these metamodel adaptations. In addition, a migrator implemented as a model transformation does not allow for the reuse of recurring migration knowledge.

Hence, adequate tool support is required to further reduce the effort involved in migrator implementation. We have performed an empirical study on the histories of two industrial metamodels to determine the requirements for adequate

tool support [7]. The study showed that there is a large fraction of recurring migration knowledge. Hence, effort can be saved by enabling the *reuse* of such recurring coupled evolution steps. However, it also revealed that there are a number of migrations that are specific to a certain domain and thus cannot be reused. In addition, the specification of these migrations requires an *expressive* language.

Currently, to our best knowledge, there is no approach that combines both the desired level of reuse and expressiveness. To alleviate this, we present COPE, an integrated approach to model the coupled evolution of metamodels and models. COPE is based on a language that provides means to combine metamodel adaptation and model migration into so-called *coupled transactions*. The stated requirements are fulfilled by two kinds of coupled transactions: reusable and custom coupled transactions. A *reusable coupled transaction* allows the reuse of recurring coupled transformations across metamodels. A *custom coupled transaction* can be manually defined by the metamodel developer for complex migrations that are specific to a metamodel. To ease the application of this language, COPE provides further abstraction by tool support. In order not to disturb the habits of the metamodel developer, we have integrated COPE into the metamodel editor. The user interface provides easy access to a number of reusable coupled transactions available through a *library*. A *language history* automatically keeps track of the consecutively performed coupled transactions.

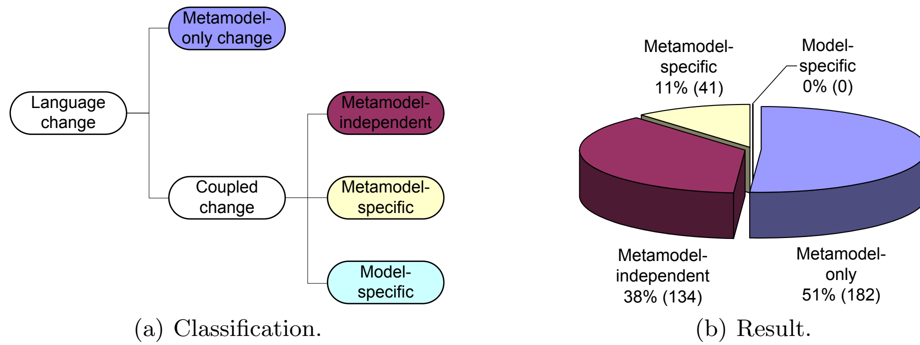
Outline. In Section 2, we recapitulate the requirements derived from our empirical study. We analyze how these requirements are fulfilled by related work in Section 3. In Section 4, we introduce the language and show how its concepts directly fulfill the requirements from the study. The seamless integration of COPE into the Eclipse Modeling Framework (EMF) is presented in Section 5. In Section 6, we show the applicability of COPE in practice by performing the coupled evolution of existing metamodels and their models. We conclude and present directions for future work in Section 7.

2 Requirements for Automated Coupled Evolution

To better understand the nature of coupled evolution of metamodels and models in practice, we performed a study on the histories of two industrial metamodels [7]. The study's main goal was to determine substantiated requirements for tool support that is adequate for coupled evolution in practice. We investigated whether reuse of migration knowledge can significantly reduce migration effort. To this end, we developed a classification of metamodel changes with respect to the automatability of the corresponding model migration.

As is depicted in Figure 1(a), we introduced four main classes of language changes. *Metamodel-only* changes do not require the migration of models, e. g. metamodel extensions like the addition of an optional attribute, whereas *coupled changes* do. *Coupled changes* can be further subdivided into metamodel-independent, metamodel-specific and model-specific coupled changes. *Metamodel-independent* coupled changes do not depend on a specific metamodel, and thus

can be reused across metamodels. Examples are well-known object-oriented refactorings [8] like e. g. rename or extract class. *Metamodel-specific* coupled changes are so specific to a certain metamodel that they cannot be reused across metamodels. An example is the removal of composite states from a statemachine metamodel which requires to flatten the state hierarchy in the models. *Model-specific* coupled changes require information from the developer of a model during migration, and thus the migration cannot be specified in a model-independent way. Examples are metamodel refinements which require to also refine the model.



(a) Classification.

(b) Result.

Fig. 1. Empirical study.

For our study in [7], we deliberately chose two metamodel histories where the impact on the models was not taken into account during metamodel adaptation. The combined result of the study for both metamodel histories is shown in Figure 1(b) as a pie chart. The figure shows the fraction and the accumulated numbers of language changes that fall into each class. As only half of the changes were metamodel-only, a significant number of language changes required a migration of existing models. As we found no model-specific coupled changes, we would have been able to specify transformations to automate the migration of all models. To this end, we do not take model-specific coupled changes into account in the following. The proportion between metamodel-independent and metamodel-specific coupled changes leads to the following two central requirements for adequate tool support:

Reuse: More than three quarters of the coupled changes were metamodel-independent, thus indicating a high potential for reuse. To take advantage of these reuse opportunities, reuse of recurring migration knowledge is required.

Expressiveness: The remaining quarter of the coupled changes were metamodel-specific and therefore required a custom model migration. Tool support automating coupled evolution must thus be sufficiently expressive to cater for complex migrations involved in metamodel-specific coupled changes.

3 Related Work

When a specification is adapted, potentially all existing instances have to be migrated in order to reconcile them with the new version of the specification. Since this problem of *coupled evolution* [9] affects all specification formalisms (e. g. database schemata, formats, grammars, metamodels) alike, numerous approaches for *coupled transformation* [10] of a specification and its instances have been proposed [11–24]. The idea of coupled transformation has even been generalized to a domain-independent approach that can be instantiated on different domains [25]. Apart from the target specification formalism, existing approaches mainly differ in their support for reuse and expressiveness. In this section, we outline approaches to coupled evolution from different domains, namely schema, grammar, format and metamodel evolution, focusing on how they fulfill the requirements rather than on idiosyncrasies of their target specification formalism. There are a number of other domains, like e. g. framework, workflow and ontology evolution, in which similar approaches are proposed.

Schema evolution denotes the migration of database instance data to an adapted version of the database schema. Schema evolution has been a field of study for several decades, yielding a substantial body of research [26, 27]. For the ORION database system, Banerjee et al. propose a fixed set of change primitives that perform coupled evolution of the schema and data [11]. While reusing migration knowledge in case of these primitives, 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 migration for the O_2 database system [12]. While more expressive, their approach does not allow for reuse of coupled transformation knowledge. In order to reuse recurring coupled transformations, SERF – as proposed by Claypool et al. – offers a mechanism to define arbitrary new high-level primitives [13], providing both reuse and expressiveness. In a nutshell, the history of approaches for schema evolution exhibits a progression towards more expressiveness and reuse. In order to fulfill the requirements from [7], COPE transfers the concepts of SERF to the domain of metamodel evolution.

Grammar evolution denotes the migration of textual programs to adaptations of their underlying grammar. Grammar evolution has been studied in the context of grammar engineering [28]. Lämmel proposes a comprehensive suite of grammar transformation operations for the incremental adaptation of context free grammars [14]. The proposed operations are based on sound, formal preservation properties that allow reasoning about the relationship between grammars before and after transformation, thus helping developers to maintain consistency of their grammar. However, the proposed operations are not coupled since they do not take the migration of words into account. Building on Lämmel's work, Pizka and Juergens propose a tool for the evolutionary development of textual languages called Lever, which is also able to automate the migration of words [15]. Primitive grammar and word evolution operations can be invoked from within a general-purpose language to perform all kinds of coupled transformation. Similar

to SERF, Lever provides a mechanism to define arbitrary new high-level primitives. COPE is not only strongly related to Lever because of its support for reuse and expressiveness, but also because it provides an explicit language history that allows to defer model migration to a later instant.

Format evolution denotes the migration of a class of documents to adaptations of their document schema. Lämmel and Lohmann suggest operators for format transformation, from which migrating transformations for documents are induced [16]. The suggested operators are based on Lämmel's work on grammar adaptation. Furthermore, Su et al. propose a complete, minimal and sound set of evolution primitives for formats and documents, and show that they preserve validity and well-formedness of both formats and documents [17]. Even though both approaches are able to automate document migration for a fixed set of format changes, they are not able to handle arbitrary, complex migrations.

Metamodel evolution denotes the migration of models in response to adaptations of their metamodel. In order to specify the model migration between two metamodel versions, Sprinkle introduces a visual graph-transformation-based language [18, 19]. Compared to conventional languages for model transformation, this language allows to specify the differences between two metamodels rather than their similarities. However, Sprinkle's language does not provide a mechanism for reusing recurring migration knowledge.

There are a number of approaches to automatically derive a model migration from the difference between two metamodel versions. Gruschko et al. classify primitive metamodel changes into non-breaking, breaking resolvable and unresolvable changes [20, 21]. Based on this classification, they propose to automatically derive a migration for non-breaking and resolvable changes, and envision to support the developer in specifying a migration for unresolvable changes. Cichetti et al. go even one step further and try to detect composite changes like e. g. extract class based on the difference between metamodel versions [22]. However, their approach is no longer automatic for composite changes which depend on each other. Although fully automated to some degree, the difference-based approaches have the disadvantage that the derived migration may not be the one intended by the developer. As a consequence, the developer has to manually modify and therefore understand the derived migration.

To avoid this problem, incremental transformation allows to capture the intention while performing metamodel adaptation. Several approaches to perform an incremental coupled transformation of metamodel and model have been proposed. Hößler and Soden present a number of high-level transformations which adapt the metamodel and migrate models [23]. These transformations are based upon a generic instance model for both metamodel and model which is required to support versioning. Wachsmuth adopts ideas from grammar engineering and proposes a classification of metamodel changes based on instance preservation properties [24]. Based on these preservation properties, the author defines a set of high-level coupled transformations. While both approaches enable the reuse

of migration knowledge, they do not provide sufficient expressiveness to cater for complex coupled transformations.

In a nutshell, there is no approach for metamodel evolution that combines both the desired level of reuse and expressiveness. To alleviate this, we propose COPE, which integrates a number of features of existing approaches. Like Sprinkle's language, COPE also relieves the metamodel developer from specifying identity rules for metamodel elements which do not have changed. COPE achieves this by using a generic instance model during migration similar to the proposal of Hößler and Soden. Based on this generic instance model, COPE follows an incremental transformation approach which allows to capture the intention while performing the metamodel adaptation. In addition, the incremental approach allows to better modularize the coupled evolution into manageable transformations, and thus to easily combine reuse with expressiveness.

4 Coupled Evolution of Metamodels and Models

In this section, we present COPE's language to specify the coupled evolution of metamodels and models. This language provides concepts to fulfill both requirements presented in Section 2: reuse of recurring migration knowledge and expressiveness to cater for domain-specific migrations. Reuse is provided by an abstraction mechanism that allows to encapsulate both metamodel adaptation and model migration in a metamodel-independent way. Expressiveness is provided by embedding primitives for metamodel adaptation and model migration into a Turing-complete language. From our experience, developers prefer to use the metamodel editor over specifying the coupled evolution in this language. Consequently, COPE provides further abstraction from this language by a non-invasive integration into a metamodel editor. For simplicity of presentation, we outline the language here, and present the tool support in Section 5.

Running example. Throughout this section, we use a statemachine metamodel as a running example. Figure 2 shows the metamodel before and after adaptation as a UML class diagram. In version 0 of the metamodel, a State has a name and may be decomposed into sub states through its subclass CompositeState. A Transition belongs to its source state and refers to a target state, and is activated by a trigger. When a state is entered, a sequence of actions is performed as effect, and in case of a composite state, an initial state is entered. For version 1 of the metamodel, the following adaptations are performed[3]:

1. The statemachine is changed from a Moore to a Mealy machine. In Moore machines, the effect of the statemachine only depends on the current state. In contrast, the effect of the statemachine depends also on the trigger in Mealy machines. Therefore, we move the attribute effect from State to Transition.
2. Regions are introduced to support concurrency within states. Therefore, we insert the class Region. We further introduce the new composition region so

[3] In Figure 1, the differences are indicated by numbered, dashed boxes.

that a composite state can define a number of concurrent regions. Finally, we move the composition state and the association initial to the new class Region, as regions are now composed of sub states.

In the following, we subsequently specify the coupled evolution in COPE's language in order to be able to migrate existing models.

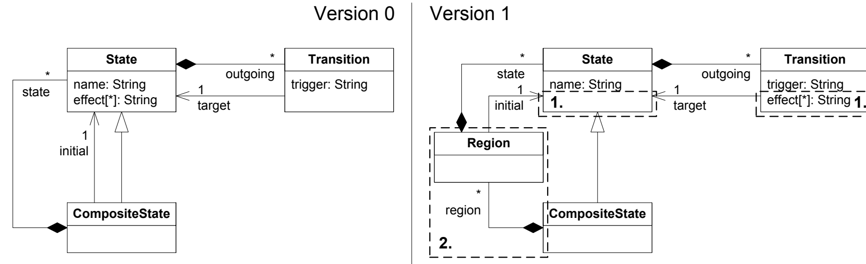


Fig. 2. Running example adaptation.

4.1 Incremental Coupled Evolution

In practice, a modeling language is evolved by incremental adaptations to the metamodel. There are a number of primitive metamodel changes like create element, rename element, delete element, and so on. One or more such primitive changes compose a specific metamodel adaptation, like in our example the introduction of regions. COPE allows to attach information about how to migrate corresponding models in response to a metamodel adaptation. Consequently, the intended model migration can already be captured while adapting the metamodel, thus preventing the loss of intention. In COPE, such a combination of metamodel adaptation and model migration is called *coupled transaction*.

Coupled transactions can be easily composed by simply sequencing them. They are modular in the sense that the corresponding model migration can be specified independently of any neighboring coupled transaction. Due to their modularity, a comprehensive evolution can be decomposed into manageable coupled transactions, thus ensuring scalability. The notion of coupled transaction qualifies to fulfill the requirements of reuse and expressiveness. Certain coupled transactions can be reused resulting in *reusable coupled transactions*, while others have to be specified manually resulting in *custom coupled transactions*.

Figure 3 illustrates how coupled transactions can be used to compose the coupled evolution of our running example. The first coupled transaction changes the statemachine metamodel from a Moore to a Mealy machine. As the corresponding model migration is specific to the metamodel, it has to be performed by a custom coupled transaction. The last two coupled transactions introduce concurrent regions to the metamodel and are invocations of reusable coupled

transactions. The invocation of ExtractClass extracts the sub states including the initial state of a composite state into the new class Region. The invocation of GeneralizeReference generalizes the multiplicity of the new reference from CompositeState to Region to enable concurrent regions.

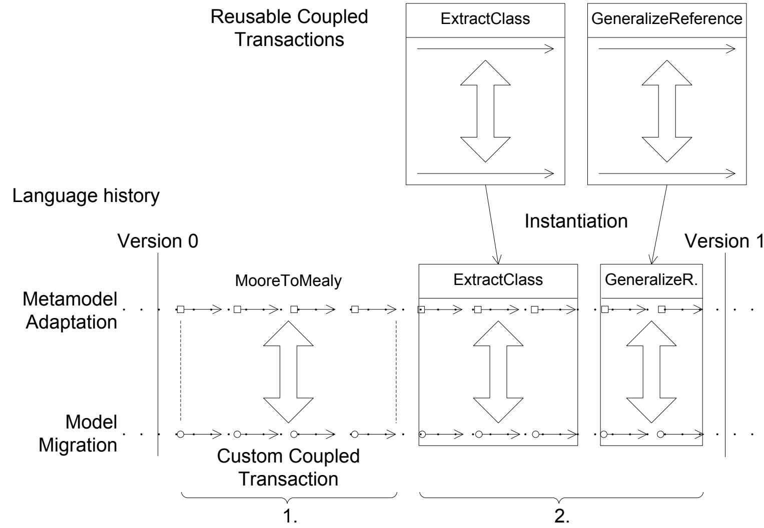


Fig. 3. Language history for the running example.

Keeping track of the coupled transactions that lead from one metamodel version to the next results in a *language history*. The language history contains enough information to migrate a model from the metamodel version to which it conforms to any subsequent metamodel version. Hence, it is particularly suited to migrate models which are not accessible while performing the metamodel adaptation. This is the case when the modeling language and the models are developed by different distributed parties. Figure 3 indicates the language history for our running example which consists of the sequence of coupled transactions together with markers for the different versions.

4.2 Coupled Transactions

Usually, the metamodel adaptation is manually performed in the metamodeling tool used for authoring the metamodel. The model migration can be manually encoded as a model transformation which transforms the old model to a new model conforming to the adapted metamodel. In general, we distinguish between *exogenous* and *endogenous* model transformation, depending on whether source and target metamodel of the transformation are different or not [29]. Exogenous model transformation requires to specify the mapping of all elements from the

source to the target metamodel. As typically only a subset of metamodel elements are modified by the metamodel adaptation, a model migration specified as an exogenous transformation contains a high fraction of identity rules. Concerning this aspect, endogenous transformation is better suited to the nature of model migration, as it only has to address those metamodel elements for which the model needs to be modified. However, endogenous transformation requires the source and the target metamodel to be the same which is not the case for metamodel evolution. Hence, conventional languages for model transformation are not well suited to specify a model migration.

Instead, model migration is best served by a language that allows to directly combine the properties of both 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 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 adapted metamodel. As a consequence, only the differences need to be specified for both metamodel adaptation and model migration. However, softening the conformance during model migration comes at the price that a model may not always conform to its metamodel. In order to ensure conformance after a certain change to metamodel and model, we require a coupled transaction to enforce the following properties:

Consistency preservation: The adapted 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 adapted 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.

We have implemented COPE on top of the Eclipse Modeling Framework (EMF) [30] which is one of the most widely used metamodeling tools. In this implementation, the conformance is softened by a generic instance model which is only used during migration. To specify both metamodel adaptation and model migration, COPE provides a number of expressive primitives which operate on the generic instance model. These primitives can be invoked from within the general-purpose scripting language Groovy [31] in order to take advantage of its expressiveness. For more information about the generic instance model and a complete list of the primitives, we refer the reader to [32].

4.3 Custom Coupled Transactions

Expressiveness is provided by custom coupled transactions, which have to be specified manually by the metamodel developer. In doing so, the metamodel developer can apply a number of primitives for both metamodel adaptation and

model migration. The primitives are complete in the sense that every possible metamodel adaptation as well as model migration can be specified with them. Completeness can be shown by first destroying the source metamodel or model, and then rebuilding the target metamodel or model from scratch as done in [11] for database schema evolution. As these primitives are embedded into the Turing-complete scripting language Groovy, the resulting language is expressive enough to even cater for very specific model migrations.

Example. Listing 1 shows the custom coupled transaction that was performed to change the statemachine from a Moore to a Mealy machine. More specifically, the depicted custom coupled transaction consists of a metamodel adaptation and a reconciling model migration. This example also shows that we only have to specify the differences for both metamodel and model in this language.

Listing 1. Custom coupled transaction MooreToMealy.

```
// metamodel adaptation
def effectAttribute = State.effect
Transition.eStructuralFeatures.add(effectAttribute)

// model migration
getEffect = { transition ->
  def effect = []
  def state = transition.target
  effect.addAll(state.get(effectAttribute))
  while(state.instanceOf(CompositeState)) {
    effect.addAll(state.initial.get(effectAttribute))
    state = state.initial
  }
  return effect
}

for(transition in Transition.allInstances) {
  def effect = getEffect(transition)
  transition.effect = effect
}

for(state in State.allInstances) {
  state.unset(effectAttribute)
}
```

The metamodel adaptation only moves the attribute effect from class State to class Transition. The attribute is assigned to the variable effectAttribute in order to be able to access its values for states, even though the attribute is no longer known to the class State. Note how metamodel elements can be accessed by means of fully qualified names (e. g. State.effect).

A Moore machine is migrated to a Mealy machine by moving the effect of each state to its incoming transitions. However, in the advent of composite states as well as initial states, the model migration is more involved. When a statemachine transitions to a composite state, it not only enters the composite state but also its initial state. Consequently, we also have to take the effect of the initial state into account when calculating the effect of the transition. Note that this may have to be applied recursively, as the initial state may again be a composite state, and so on. The model migration encoded in COPE's language is thus divided into two passes: First, we set the effect for each transition based on the states, and then we remove the effect from each state. The language provides the primitive allInstances to be able to iterate over all instances of a certain type. The effect of a transition is set by using transition.effect = effect which is a short form for transition.set(Transition.effect, effect). The effect of a transition is calculated by means of the helper method getEffect. As explained before, the effect consists of the effect of the transition's target state as well as the effects of the initial states. transition.target is the short form for transition.get(Transition.target). However, the short forms can only be used, in case a feature of that name is currently defined by the instance's type. As the attribute effect is no longer defined for class State, we thus have to use state.get(effectAttribute) to be able to access the effect of a state. Furthermore, the primitive instanceOf can be used to check whether a state is of type CompositeState. The effect of a state is removed by using a primitive to unset the effectAttribute.

4.4 Reusable Coupled Transactions

Reuse is provided by an abstraction mechanism to generalize coupled transactions into so-called reusable coupled transactions. Reusable coupled transactions are specified independently of the metamodel, and encapsulate both metamodel adaptation and reconciling model migration. They can be reused across metamodels, thus promising to significantly reduce effort associated with metamodel adaptation and model migration. COPE allows to declare new reusable coupled transactions and make them available through a *library*. The language employs the abstraction mechanism of procedures in Groovy in order to declare reusable coupled transactions. A reusable coupled transaction is declared independently of the specific metamodel by means of parameters. Reusable coupled transactions can be instantiated by invoking the procedure with parameters assigned to specific metamodel elements. The applicability of a reusable coupled transaction can be restricted by preconditions in the form of assertions.

Example. Listing 2 shows the invocation of the reusable coupled transactions ExtractClass and GeneralizeReference, which correspond to the second adaptation in our example history. ExtractClass is invoked to extract the references state and initial from CompositeState to the new class Region. The extracted region then is accessible from a composite state through the new single-valued containment reference named region. GeneralizeReference is invoked to increase the multiplicity of this new reference in order to enable multiple concurrent regions. Note that

by invoking reusable coupled transactions, the metamodel developer does not have to specify neither metamodel adaptation nor model migration.

Listing 2. Instantiation of reusable coupled transactions.

```
extractClass([CompositeState.state, CompositeState.initial],
     "Region", "region")
generalizeReference(CompositeState.region, Region, 1, INF)
```

Listing 3 shows the declaration of the reusable coupled transaction ExtractClass which we just invoked to introduce regions into our example metamodel. This reusable coupled transaction extracts a number of features from a context class to a new class. The extracted class is accessible from the context class through a new single-valued containment reference. The reusable coupled transaction declares parameters for the attributes and references to be extracted (features), the name of the new class (className) and the name of the new reference (referenceName). Several preconditions in the form of assertions restrict the applicability of the reusable coupled transaction, e. g. every feature has to belong to the same context class.

The metamodel adaptation creates the extracted class and the new single-valued containment reference from the context class to the extracted class. Then, the extracted features are moved from the context class to the extracted class. For the metamodel adaptation, we use the primitives of the meta-metamodel implementation together with some high-level primitives to create new metamodel elements (e. g. newEClass). The reusable coupled transaction is simplified in the sense that it leaves out the package in which the extracted class is created.

The model migration pretty much modifies the model accordingly. For each instance of the context class (contextInstance), a new instance of the extracted class is created and associated to the context instance through the new reference. Then, all the values of the extracted features are moved from the context instance to the new instance. Note that due to the generic instance model the context instance's value of a feature can still be accessed by the unset method, even though the feature has already been moved to the extracted class.

5 Tool Support

From our experience, metamodel developers do not want to script the coupled evolution, but rather prefer to adapt the metamodel directly in an editor. Consequently, COPE is implemented as a non-invasive integration into the existing EMF metamodel editor. Even though COPE is based on the language presented in Section 4, it shields the metamodel developer from this language as far as possible. COPE is open source and can be obtained from our website[4]. The web

[4] http://cope.in.tum.de

Listing 3. Declaration of reusable coupled transaction ExtractClass.

```
extractClass = {List<EStructuralFeature> features, String
    className, String referenceName ->

  def EClass contextClass = features[0].eContainingClass

  // preconditions
  assert features.every{feature -> feature.eContainingClass ==
      contextClass} :
    "The features have to belong to the same class"
  assert contextClass.getEStructuralFeature(referenceName) ==
      null || features.contains(contextClass.
      getEStructuralFeature(referenceName)) :
    "A feature with the same name already exists"

  // metamodel adaptation
  def extractedClass = newEClass(className)
  def reference = contextClass.newEReference(referenceName,
      extractedClass, 1, 1, CONTAINMENT)
  extractedClass.eStructuralFeatures.addAll(features)

  // model migration
  for(contextInstance in contextClass.allInstances) {
    def extractedInstance = extractedClass.newInstance()
    contextInstance.set(reference, extractedInstance)
    for(feature in features) {
      extractedInstance.set(feature, contextInstance.unset(
          feature))
    }
  }
}
```

site also provides a screencast, documentation and several examples (including the running example from this paper). We first describe the workflow that is supported by COPE, before detailing on its integration into the user interface.

5.1 Tool Workflow

Figure 4 illustrates the tool workflow using the running example from Section 4.

COPE provides a *library* of reusable coupled transactions that can be invoked on a specific metamodel. Besides the transactions used in Section 4, the current library contains a number of other reusable coupled transactions like e. g. Rename or DeleteFeature. The library is extensible in the sense that new reusable coupled operations can be declared and registered. Reusable coupled transactions are declared independently of the specific metamodel, i. e. on the level of the meta-metamodel.

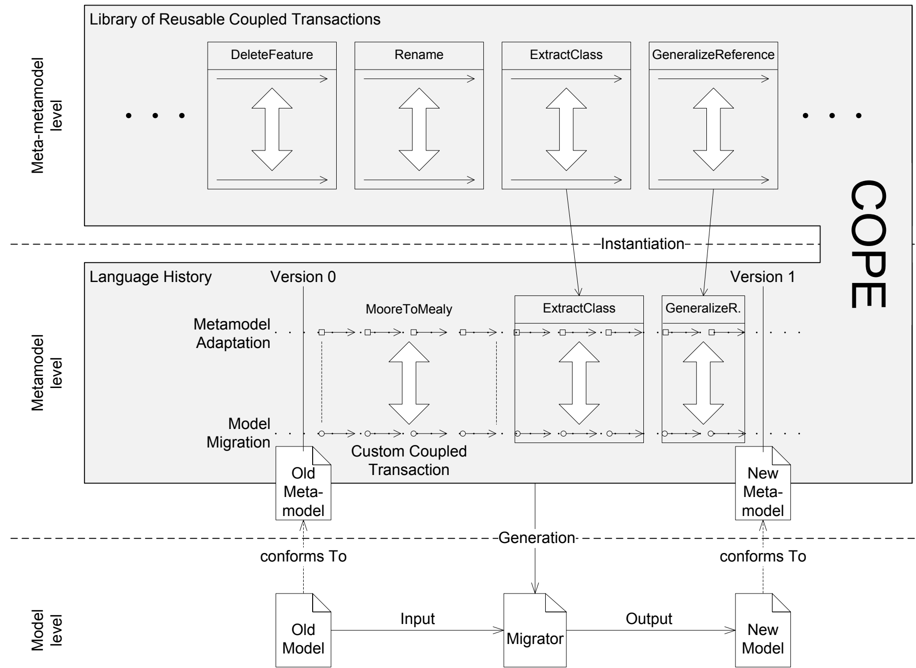


Fig. 4. Tool workflow.

All changes performed to the metamodel are maintained in an explicit *language history*. The history keeps track of the coupled transactions which contain both metamodel adaptation and model migration. It 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 4, the evolution from version 0 to version 1 is the sequence of coupled transactions we performed in Section 4.

A *migrator* can be generated from the language history that allows for the batch migration of models. The migrator can be invoked to automatically migrate existing models, i. e. no user interaction is required during migration.

5.2 User Interface

Figure 5 shows an annotated screen shot of COPE's user interface. COPE has been integrated into the existing structural metamodel editor provided by EMF (a). This metamodel editor has been extended so that it also provides access to the language history (b). Reusable coupled transactions are made available to the metamodel developer through a special view called *operation browser* (c). An editor with syntax highlighting is provided for the specification of custom coupled transactions (d).

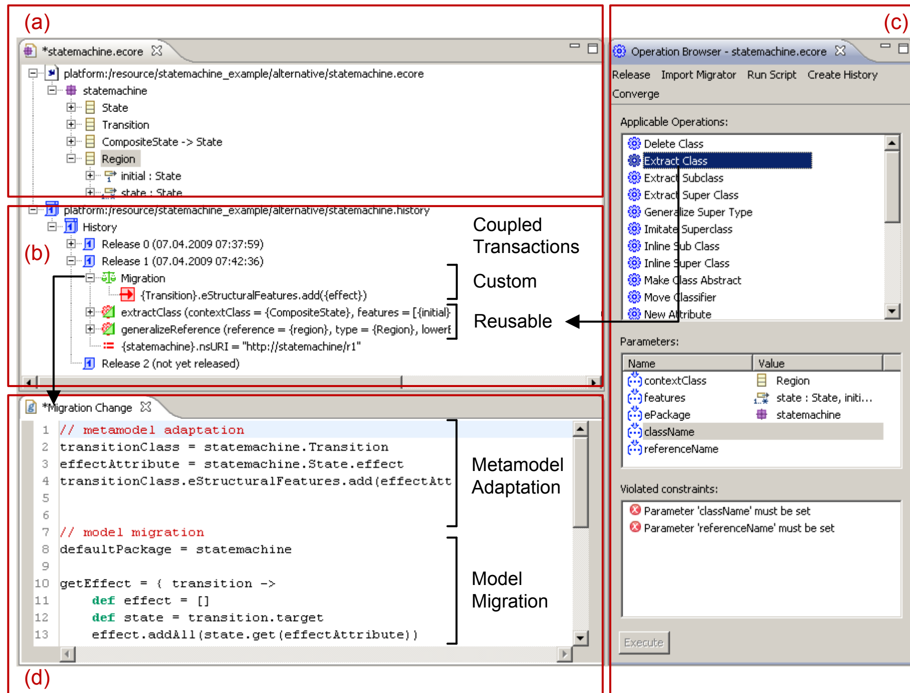


Fig. 5. Integration of COPE into the EMF metamodel editor.

The metamodel developer can adapt the metamodel by invoking reusable coupled transactions through the *operation browser*. The browser is context-sensitive, i. e. offers only those reusable coupled transactions that are applicable to the elements currently selected in the metamodel editor. The operation browser allows to set the parameters of a reusable coupled transaction based on their type, and gives feedback on its applicability based on the preconditions. When a reusable coupled transaction is executed, its invocation is automatically tracked in the language history. Figure 5 shows the ExtractClass operation being available in the browser (c), and the reusable coupled transactions stored in the history (b). Note that the metamodel developer does not have to know about the coupled evolution language if she is only invoking reusable coupled transactions.

In case no reusable coupled transaction is available for the coupled evolution at hand, the metamodel developer can perform a custom coupled transaction. First, the metamodel is directly adapted in the metamodel editor, in response to which the changes are automatically tracked 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. Note that the metamodel adaptation is automatically generated from the changes tracked in the history. In order to allow for different metamodeling habits, adapting the metamodel and attaching a model migration is temporally decoupled such that a model migration can be attached at any later

instant. Figure 5 shows the model migration attached to the manual changes (b) in a separate editor with syntax highlighting (d).

The operation browser provides a release button to create a major version of the metamodel. After release, the metamodel developer can initiate the automatic generation of a migrator.

6 Case Study

In order to demonstrate its applicability in practice, we used COPE to model the coupled evolution of two existing metamodels. The detailed results of the case study in the form of the language histories as presented in Section 5 can be obtained from our website[5].

6.1 Goals

The study was performed to test the applicability of COPE to real-world coupled evolution and better understand the potential for reuse of recurring migration knowledge. More specifically, the study was performed to answer the following research questions:

- Which fraction of the changes are simple metamodel extensions that do trivially not require a migration of models?
- Which fraction of the changes can be reused by means of reusable coupled transactions?
- Which fraction of the changes have to be implemented by means of custom coupled transactions?
- Can COPE be applied to specify the complete coupled evolution of real-world metamodels, i. e. including all intermediate versions?

6.2 Setup

As input to our study, we chose two EMF-based metamodels that already have an extensive evolution history. We deliberately chose metamodels from completely different backgrounds in order to achieve more representative results.

The first metamodel is developed as part of the open source project Graphical Modeling Framework[6] (GMF). It is used to define generator models from which code for a graphical editor is generated. For our case study, we modeled the coupled evolution from release 1.0 over 2.0 to release 2.1, which covers a period of 2 years. There exist a significant number of models conforming to this metamodel, most of which are not under control of the developers. In order to be able to migrate these models, the developers have handcrafted a migrator with test cases which can be used for validation.

[5] `http://cope.in.tum.de`
[6] `http://www.eclipse.org/modeling/gmf`

The second metamodel is developed as part of the research project Palladio Component Model[7] (PCM), and is used for the specification and analysis of component-based software architectures. For our case study, we modeled the coupled evolution from release 2.0 over 3.0 to release 4.0, which covers a period of 1.5 years. As the metamodel developers control the few models, they were not forced to handcraft a migrator until now, but manually migrated the models instead. Since no migrator could be used for validation for this reason, the modeled coupled evolution was validated by the developers of PCM.

The evolution of the metamodels was only available in the form of snapshots that depict the state of the metamodel at a particular point in time. To this end, we had to infer both the metamodel adaptation as well as the corresponding model migration. We used the following systematic procedure to reverse engineer the coupled evolution:

1. *Extraction of metamodel versions*: We extracted versions of the metamodel from the version control system.
2. *Comparison of subsequent metamodel versions*: Since the version control systems of both projects are snapshot-based, they provide no information about the differences between the metamodel versions. Therefore, successive metamodel versions had to be compared to obtain a difference model. The difference model consists of a number of primitive changes between subsequent metamodel versions and was obtained by means of tool support[8].
3. *Generation of metamodel adaptation*: A first version of the history was obtained by generating a metamodel adaptation from the difference model between subsequent metamodel versions. For this purpose, a transformation was implemented that translates each of the primitive changes from the difference model to metamodel adaptation primitives specified in COPE.
4. *Detection of coupled transactions*: The generated metamodel adaptation was refined by combining adaptation primitives to coupled transactions based on the information on how corresponding models are migrated. In doing so, we always tried to map the compound changes to reusable coupled transactions already available in the library. If not possible, we tried to identify and develop new reusable coupled transactions. In case a certain model migration was too specific to be reused, it was realized as a custom coupled transaction.
5. *Validation of the history*: The validity of the obtained coupled evolution was tested on both levels. The metamodel adaptation is easy to validate, because the history can be executed and the result can be compared to the metamodel snapshots. Test models before and after model migration were used to validate whether the model migration performs as intended.

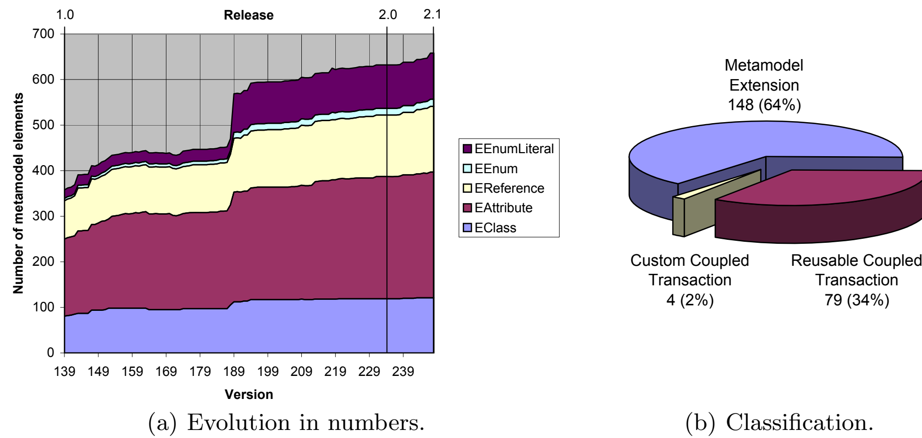
Steps 1 to 3 as well as 5 are fully automated, whereas step 4 had to be performed manually. In addition, there is an iteration over steps 4 and 5, as a failed validation leads to corrections of the history. It took roughly one person week for each studied metamodel to reach the fix point during the iteration. However,

[7] `http://www.palladio-approach.net`
[8] `http://wiki.eclipse.org/index.php/EMF_Compare`

in this case study, the coupled evolution was obtained by reverse engineering, which requires a lot of effort for understanding the intended migration. We are convinced that the metamodel developers can model the coupled evolution with significantly less effort, when they use COPE for forward engineering.

6.3 Results

As the GMF developers do not have all the models under control, they employ a systematic change management process: the developers discuss metamodel adaptations and their impact on models thoroughly before actually carrying them out. Consequently, we found no destructive change at any instant in the history, that was reversed at a later instant. To this end, the obtained language history comprises all the intermediate versions. Figure 6(a) gives an impression of the size of the studied metamodel and its evolution over all the metamodel versions. In addition, the figure indicates the different releases of the metamodel. The metamodel is quite extensive, accumulating more than a hundred classes in the course of its history.



(a) Evolution in numbers.

(b) Classification.

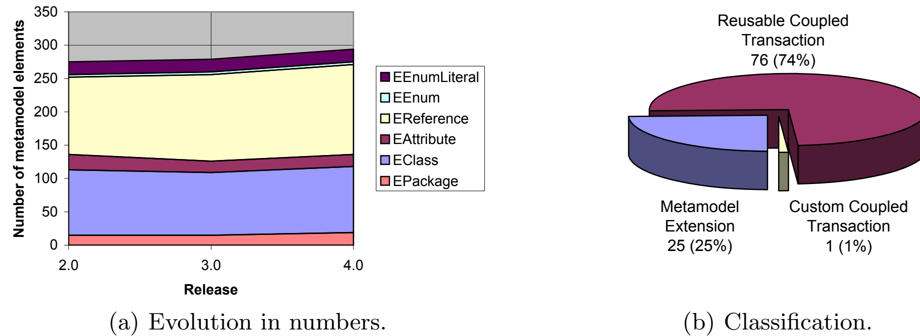
Fig. 6. History of the GMF metamodel.

Figure 6(b) depicts the number of the different classes of metamodel adaptations that were used to model the coupled evolution using COPE. The metamodel extensions make up 64% of the adaptations, whereas reusable coupled transactions account for 34%. Table 1 refines this classification by listing the names and the number of occurrences of the different kinds of metamodel adaptations. The dashed line distinguishes the reusable coupled transactions known from the literature from those which have been implemented while conducting the case study. For the GMF metamodel, these new reusable coupled transactions cover 15 out of 79 occurrences (19%). The remaining 2% of the metamodel adaptations

consist of only 4 custom coupled transactions for which the model migration had to be implemented manually. The model migration code handcrafted for these custom coupled transactions amounts to 100 lines of code.

As the developers of the GMF metamodel do not have all the models under their control, they have manually implemented a migrator. This migrator constitutes a very technical solution, and is based on different mechanisms for the two stages. For the migration from release 1.0 to 2.0, the migrator patches the model while deserializing its XML representation. For the migration from release 2.0 to 2.1, a generic copy mechanism is used that first filters out non-conforming parts of the model, and later rebuilds them. Even though this migrator is very optimized, it is difficult to understand and maintain due the low abstraction level of its implementation.

As the developers of the PCM metamodel have all the models under their control, they apparently have not taken the impact on the models into account. Consequently, there were a lot of destructive changes between the intermediate versions, that were reversed at a later instant. To this end, the obtained language history comprises only the release versions. Figure 7(a) gives an impression of the size of the metamodel and its evolution over the studied releases. Similar to GMF, the PCM metamodel is quite extensive, being split up in a number of packages and defining more than a hundred classes throughout the history.



(a) Evolution in numbers.

(b) Classification.

Fig. 7. History of the PCM metamodel.

Figure 7(b) depicts the number of the different classes of metamodel adaptations that were used to model the coupled evolution. Here, the metamodel extensions account for only 25% of the metamodel adaptations, whereas reusable coupled transactions make up 74%. Again, Table 1 provides more detailed results. The reusable coupled transactions that were implemented while conducting the case study cover 12 out of 76 occurrences (16%). The remaining 1% of the metamodel adaptations consist of 1 custom coupled transaction for which the model migration had to be implemented manually. The model migration code handcrafted for this custom coupled transaction amounts to only 10 lines of code.

As the developers have not yet provided tool support for model migration, our approach helped by providing an automatic migrator. However, they provided us with test models and helped to validate the obtained model migration.

	GMF			PCM			Overall
	1.0 - 2.0	2.0 - 2.1	Overall	2.0 - 3.0	3.0 - 4.0	Overall	
Metamodel Extension	136	12	148	9	16	25	173
Add Super Type				1	2	3	3
New Attribute	63	6	69		1	1	70
New Class	36	1	37	3	4	7	44
New Enumeration	12	1	13		4	4	17
New Package					4	4	4
New Reference	25	4	29	5	1	6	35
Reusable Coupled Transaction	76	3	79	44	32	76	155
Change Attribute Type				2	1	3	3
Delete Class				1		1	1
Delete Feature	14		14	4	2	6	20
Extract Class	1		1				1
Extract Super Class	5		5				5
Generalize Reference	5		5	2	2	4	9
Generalize Super Type				1	2	3	3
Inline Super Class	2		2				2
Move Classifier				1	4	5	5
Move Feature	2	1	3				3
Pull up Feature	3		3				3
Push down Feature	1		1				1
Rename	27	1	28	16	18	34	62
Replace Inheritance by Delegation	1	1	2	4		4	6
Specialize Super Type				3	1	4	4
Collect Feature	4		4				4
Combine Feature	1		1				1
Copy Feature	1		1				1
Extract and Group Attribute	1		1				1
Extract existing Class				2		2	2
Flatten Hierarchy	1		1				1
Propagate Feature	1		1				1
Remove Superfluous Super Type					1	1	1
Remove Super Type	1		1		1	1	2
Replace Class	2		2	7		7	9
Replace Enumeration	2		2				2
Replace Literal	1		1				1
Specialize Composition				1		1	1
Custom Coupled Transaction	2	2	4	1		1	5

Table 1. Detailed results.

6.4 Discussion

The fraction of metamodel extensions is very large for the GMF metamodel, whereas it is rather small for the PCM metamodel. A possible interpretation is that the GMF developers were as far as possible avoiding metamodel adaptations that required to enhance the migrator. The reason for the metamodel extensions could as well be the nature of the evolution: they were adding new generator features to the language which are orthogonal to existing ones.

For both metamodels, a large fraction of changes can be dealt with by reusable coupled transactions – aside from the metamodel extensions. This result strengthens the findings from the previous study as presented in Section 2 that a lot of migration effort can be saved by reuse in practice. Besides the reusable

coupled transactions known from the literature, we have also identified a number of new reusable coupled transactions. It may seem odd that these new reusable coupled transactions could be used for one metamodel, but not for the other. However, two case studies may not suffice to show their usefulness in other scenarios. In addition, it may depend on the habits of the developer which reusable coupled transactions are often used and which not. The extension mechanism of COPE allows the developer to easily register new reusable coupled transactions which fit their habits.

For both metamodels, a very small fraction of changes were so specific that they had to be modeled as custom coupled transactions. Due to the expressiveness of the language, it was not difficult to manually implement these custom coupled transactions. This result also strengthens the findings from the previous study that a non-negligible number of changes are specific to the metamodel.

The case studies further showed that COPE can be applied to specify the coupled evolution of real-world metamodels. In case of the GMF metamodel, we would even have been able to directly use COPE for its maintenance. As the GMF developers do not control the numerous existing models, they took also the impact on the models into account while adapting the metamodel. COPE can help here to perform more profound metamodel adaptations. In case of the PCM metamodel, we would not have been able to directly use COPE for its maintenance. For metamodel adaptation, the PCM developers preferred flexibility over preservation of existing models, as they have the few existing models under control. COPE can help here to perform the metamodel adaptations in a more systematic way by using reusable coupled transactions. Summing up, COPE provides a compromise between the two studied types of metamodel histories: its provides more flexibility for carrying out metamodel adaptations, and offers at the same time a more systematic approach for metamodel adaptation.

7 Conclusion

Just as other software artifacts, modeling languages and thus their metamodels have to be adapted. In order to reduce the effort for the resulting migration of models, adequate tool support is required. In previous work, we have performed a study on the histories of two industrial metamodels to determine requirements for adequate tool support. Adequate tool support needs to support the reuse of migration knowledge, while at the same time being expressive enough for complex migrations. To the best of our knowledge, existing approaches for model migration do not cater for both reuse and expressiveness. This paper presented COPE, an integrated approach fulfilling these requirements. Using COPE, the coupled evolution can be incrementally composed of coupled transactions that only require specification of the differences of metamodel and models in consecutive versions. The resulting modularity of coupled transactions ensures scalability, and is particularly suited to combine reuse with expressiveness. Reuse is provided by reusable coupled transactions that encapsulate recurring migration knowledge. Expressiveness is provided by a complete set of primitives embedded

into a Turing-complete language, which can be used to specify custom coupled transactions. Tracking the performed coupled transactions in an explicit language history allows to migrate models at a later instant, and provides better traceability of metamodel adaptations. We implemented these language concepts based on the Eclipse Modeling Framework (EMF). To ease its application, COPE was seamlessly integrated into the metamodel editor, shielding the metamodel developer from technical details as far as possible. We demonstrated the applicability of COPE to real-world language evolution by reproducing the coupled evolution of two existing modeling languages over several years. These case studies strengthen the findings of our previous study [7]: while reuse saves a lot of effort, expressiveness is required for the rare, but important cases of complex migrations.

Future Work. During the case studies, we have validated the usefulness of well-known reusable coupled transactions, but also identified a number of new ones. Until now, we pretty much developed new reusable coupled transactions in a demand-driven way. However, we plan to compile a library of well-tested reusable coupled transactions that cover most scenarios of metamodel evolution. To this end, the existing ones may have to be refined, consolidated and aligned more orthogonally to each other. Building on [24], we intend to classify reusable coupled transactions according to instance preservation properties so that the metamodel developer can better assess their impact on models.

Currently, conformance preservation of a coupled transaction can only be verified while executing it on a certain model. To enable the verification of conformance preservation in a model-independent way, we intend to develop a static analysis. In contrast to the verification of properties, validation is more concerned with whether the migration performs as intended. In order to validate coupled transactions, we plan to develop a framework for the rigorous testing of model migrations. This may include specific coverage criteria as well as a method to derive new test models.

In this paper, we were only concerned with the migration of models in response to metamodel adaptation. However, there are also other artifacts like e. g. editors and generators which depend on the metamodel and which thus have to be migrated. We first focused on model migration, as the number of models of a successful modeling language typically outnumbers the number of other artifacts. To this end, we intend to extend COPE in a way that also the migration of other artifacts can be specified. Especially for reusable coupled transactions, we plan an extension mechanism to allow for the injection of migration code for other artifacts.

As we already mentioned, our approach is especially suited for the incremental development and maintenance of modeling languages. We claim that a good modeling language is hard to obtain by an upfront design, but rather has to be developed by an evolutionary process. A version of a modeling language is defined and deployed to obtain feedback from its users, which again may lead to a new version. We thus plan to define a systematic process in order to support the evolutionary development of modeling languages. This process should also

cover the maintenance of existing modeling languages. To that end, it should also provide methods to identify bad metamodel designs and to replace them by better designs.

Acknowledgements. We are thankful to the PCM developers – especially Steffen Becker, Franz Brosch, and Klaus Krogmann – to grant us access to their metamodel history, and for the effort spent on migration validation. We also like to thank Steffen Becker, Antonio Cicchetti, Thomas Goldschmidt, Steven Kelly, Anneke Kleppe, Klaus Krogmann, Ed Merks, Alfonso Pierantonio, Juha-Pekka Tolvanen, Sander Vermolen, Markus Voelter, and Guido Wachsmuth for encouraging discussion, and for helpful suggestions. We are also grateful to anonymous reviewers for comments on earlier versions of this paper.

References

1. Kleppe, A.G., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley, Boston, MA, USA (2003)
2. Greenfield, J., Short, K., Cook, S., Kent, S.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley (2004)
3. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling. John Wiley & Sons (2007)
4. AUTOSAR Development Partnership: AUTOSAR Specification V3.1 (2008)
5. Favre, J.M.: Languages evolve too! changing the software time scale. In: 8th International Workshop on Principles of Software Evolution (IWPSE), IEEE Computer Society (2005) 33–44
6. Object Management Group: Unified Modeling Language, Superstructure, v2.1.2 (2007)
7. Herrmannsdoerfer, M., Benz, S., Juergens, E.: Automatability of Coupled Evolution of Metamodels and Models in Practice. In Czarnecki, K., Ober, I., Bruel, J.M., Uhl, A., Völter, M., eds.: MODELS 2008. Volume 5301 of LNCS., Springer Heidelberg (2008) 645–659
8. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley Longman Publishing Co., Inc. (1999)
9. Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., Jazayeri, M.: Challenges in software evolution. In: 8th International Workshop on Principles of Software Evolution (IWPSE). (2005) 13–22
10. Lämmel, R.: Coupled Software Transformations (Extended Abstract). In: 1st International Workshop on Software Evolution Transformations. (2004)
11. Banerjee, J., Kim, W., Kim, H.J., Korth, H.F.: Semantics and implementation of schema evolution in object-oriented databases. Volume 16., ACM (1987) 311–322
12. Ferrandina, F., Meyer, T., Zicari, R., Ferran, G., Madec, J.: Schema and database evolution in the O2 object database system. In: 21th International Conference on Very Large Data Bases (VLDB), Morgan Kaufmann (1995) 170–181
13. Claypool, K.T., Jin, J., Rundensteiner, E.A.: SERF: schema evolution through an extensible, re-usable and flexible framework. In: 7th International Conference on Information and Knowledge Management (CIKM), ACM (1998) 314–321
14. Lämmel, R.: Grammar adaptation. In: FME 2001. Volume 2021 of LNCS., Springer Heidelberg (2001) 550–570

15. Pizka, M., Juergens, E.: Automating language evolution. In: 1st Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE), IEEE Computer Society (2007) 305–315
16. Lämmel, R., Lohmann, W.: Format Evolution. In: 7th International Conference on Reverse Engineering for Information Systems (RETIS). Volume 155., OCG (2001) 113–134
17. Su, H., Kramer, D., Chen, L., Claypool, K., Rundensteiner, E.A.: XEM: Managing the Evolution of XML Documents. In: 11th International Workshop on research Issues in Data Engineering (RIDE), IEEE Computer Society (2001) 103
18. Sprinkle, J.M.: Metamodel driven model migration. PhD thesis, Nashville, TN, USA (2003)
19. Sprinkle, J., Karsai, G.: A domain-specific visual language for domain model evolution. Journal of Visual Languages and Computing **15** (2004) 291–307
20. Becker, S., Gruschko, B., Goldschmidt, T., Koziolek, H.: A Process Model and Classification Scheme for Semi-Automatic Meta-Model Evolution. In: 1st Workshop MDD, SOA und IT-Management (MSI), GI, GiTO-Verlag (2007) 35–46
21. Gruschko, B., Kolovos, D., Paige, R.: Towards synchronizing models with evolving metamodels. In: International Workshop on Model-Driven Software Evolution. (2007)
22. Cicchetti, A., Ruscio, D.D., Eramo, R., Pierantonio, A.: Automating co-evolution in model-driven engineering. In Ceballos, S., ed.: 12th International Enterprise Distributed Object Computing Conference (EDOC), IEEE Computer Society (2008)
23. Hößler, J., Soden, M., Eichler, H.: Coevolution of Models, Metamodels and Transformations. In: Models and Human Reasoning. Wissenschaft und Technik Verlag, Berlin (2005) 129–154
24. Wachsmuth, G.: Metamodel adaptation and model co-adaptation. In: ECOOP 2007. Volume 4609 of LNCS., Springer Heidelberg (2007) 600–624
25. Vermolen, S.D., Visser, E.: Heterogeneous coupled evolution of software languages. In Czarnecki, K., Ober, I., Bruel, J.M., Uhl, A., Völter, M., eds.: MODELS 2008. Volume 5301 of LNCS., Springer Heidelberg (2008) 630–644
26. Li, X.: A survey of schema evolution in object-oriented databases. In: 31st International Conference on Technology of Object-Oriented Language and Systems (TOOLS), IEEE Computer Society (1999) 362
27. Rahm, E., Bernstein, P.A.: An online bibliography on schema evolution. SIGMOD Rec. **35**(4) (2006) 30–31
28. Klint, P., Lämmel, R., Verhoef, C.: Toward an engineering discipline for grammarware. ACM Trans. Softw. Eng. Methodol. **14**(3) (2005) 331–380
29. Mens, T., Van Gorp, P.: A taxonomy of model transformation. Electronic Notes in Theoretical Computer Science **152** (2006) 125–142
30. Budinsky, F., Brodsky, S.A., Merks, E.: Eclipse Modeling Framework. Pearson Education (2003)
31. Koenig, D., Glover, A., King, P., Laforge, G., Skeet, J.: Groovy in Action. Manning Publications Co., Greenwich, CT, USA (2007)
32. Herrmannsdoerfer, M., Benz, S., Juergens, E.: COPE: A Language for the Coupled Evolution of Metamodels and Models. In: 1st International Workshop on Model Co-Evolution and Consistency Management. (2008)