Automated co-evolution of GMF editor models

Davide Di Ruscio[1], Ralf Lämmel[2], Alfonso Pierantonio[1]

[1] Computer Science Department, University of L'Aquila, Italy
[2] Software Languages Team, Universität Koblenz-Landau, Germany

Abstract. The Eclipse Graphical Modeling (GMF) Framework provides the major approach for implementing visual languages on top of the Eclipse platform. GMF relies on a family of modeling languages to describe abstract syntax, concrete syntax as well as other aspects of the visual language and its implementation in an editor. GMF uses a model-driven approach to map the different GMF models to Java code. The framework, as it stands, lacks support for evolution. In particular, there is no support for propagating changes from the domain model (i.e., the abstract syntax of the visual language) to other editor models. We analyze the resulting co-evolution challenge, and we provide a solution by means of GMF model adapters, which automate the propagation of domain-model changes. These GMF model adapters are special model-to-model transformations that are driven by difference models for domain-model changes.

1 Introduction

In the context of Model Driven Engineering (MDE) [2], the definition of a domain-specific modeling language (DSML) or its implementation in an editor (or another tool) consists of *a collection of coordinated models*. These models specify the abstract and concrete syntaxes of the language, and possibly further aspects related to semantics or the requirements of a particular DSML tool. The increasing understanding of the problem domain for DSML may necessitate continuous evolution. Hence, DSML have to evolve, and DSL tools have to co-evolve [11].

In the present paper, we make a contribution to the general theme of *evolution for DSMLs* by addressing the more specific problem of supporting co-evolution between the coordinated models that constitute the definition of a DSML, or, in fact, its implementation in a graphical editor. We focus on the *propagation of abstract-syntax changes to other models*, e.g., the model for a graphical, concrete syntax.

In MDE, the abstract syntax of DSMLs is typically expressed in terms of metamodels which are created by means of generic modeling environments, e.g., the Eclipse Modeling Framework (EMF) [9, 3]. Indeed, we leverage EMF in the present paper. Further, we build upon the Eclipse Graphical Modeling Framework (GMF) for developing graphical editors based on EMF and other Eclipse components [14]. Arguably, GMF defines the mainstream approach to graphical editor development within the Eclipse platform. GMF uses a generative approach to obtain a working editor implementation (in Java) from the coordinated models of the editor for a DSML.

For illustration, consider the simple mind-map editor in Fig. 1. We have annotated the different panes of the editor with the associated GMF models underneath. The do-

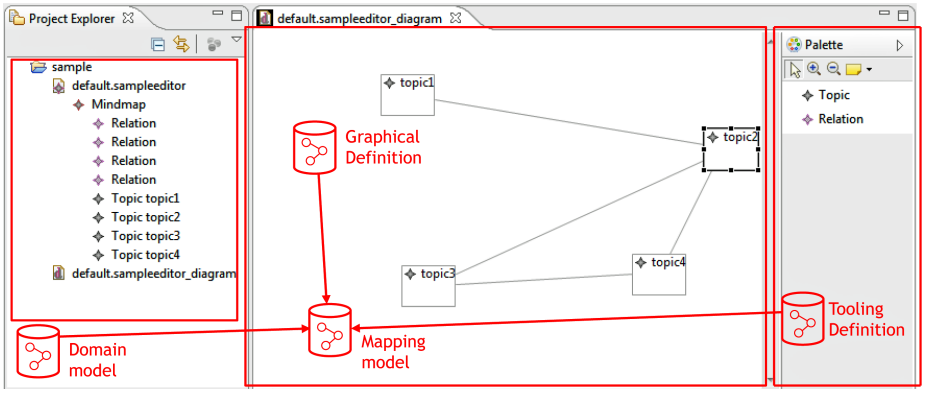

Fig. 1. Snapshot of a simple editor with indications of underlying models.

main model is concerned with the abstract syntax. The graphical and the tooling definition are concerned with concrete syntax and the editor functionality. The mapping model connects the various models. We will describe the architecture of the editor in more detail later. In the GMF approach, model-to-model and model-to-code transformations derive the implementation of a graphical editor which provides the means for editing models conforming to the specified domain model.

With such a multi-model and generative approach to editor implementation, changes to the abstract syntax (i.e., the metamodel) invalidate instances (i.e., models), other editor models, generated code, and all model transformations that may depend on the aforementioned models. In previous work, the problem of metamodel/model co-evolution has been addressed [4, 29], but the problem of co-evolution among coordinated editor models is largely unexplored. The present paper specifically contributes to this open problem, which helps with the co-evolution of DSML editors as opposed to the co-evolution of pre-existing DSML models. More specifically, we are concerned with the questions what and how GMF models need to be co-changed in reply to changes of the domain model (say, metamodel, or abstract syntax definition) of the editor. The co-evolution challenge at hand is to adapt GMF editors when changes are operated on the domain model.

The GMF framework does not support such co-evolution, and this somewhat diminishes the original goal of GMF to aggressively simplify the development of graphical editors. That is, while it is reasonable simple to draft and connect all GMF models from scratch, it is notably difficult to evolve an editor through changes of specific GMF models. A recurring focus for evolution is the domain model of the editor. When the domain model is changed, the user may notice that the editor has to be fixed through unsuccessful runs of some generator, the compiler, or the editor itself, and in all cases, subject to error messages at a low level of abstraction. Alternatively, the user may attempt to regenerate some models through the available wizards (model-to-model transformations of GMF), which however means that the original, possibly customized models are lost.

The complexity of the co-evolution problem for DSML editors has been recognized also by others (e.g., see [20, 27]). Also, in [27], the authors discuss that the GMF infrastructure has a number of limitations, some of them related to co-evolution, and they even propose an alternative solution to define graphical editors for modeling languages. Even outside the MDE context, the identified co-evolution challenge is relevant, and it has not been generally addressed. For instance, in compiler construction and domain-specific language implementation for languages with textual syntax, the same kind of breaking changes to abstract or concrete syntaxes can happen, and not even the most advanced transformation- or mapping-based approaches to the coordination of the syntaxes readily address the challenge (e.g., see [18, 30]). The challenge is only exacerbated by the multiplicity of coordinated models in an approach for DSML editors such as with GMF.

The contributions of the paper can be summarized as follows:

- We analyze GMF's characteristics in terms of the co-evolution of the various models that contribute to a GMF editor. Starting from conceived domain-model changes, their implications for the editor itself and other GMF models are identified.
- Even though catalogues for metamodel changes are available from multiple existing works (e.g., [4, 28, 29]—to mention a few), the application of such a change catalogue to a different scenario (i.e., the editor models in a "dependency" relation) is a novelty.
- We address the resulting co-evolution challenge by complementing GMF's wizard- and generator-biased architecture with GMF adapters, which are model-to-model transformations that propagate changes of the domain model to other models.
- The GMF adapters leverage on a difference model which is used to represent differences between subsequent versions of a given metamodel. Such difference models have been used in previous works on co-evolution, but the illustration of their applicability to the new kind of co-evolution challenge at hand is an important step towards their promotion to a general MDE technique.

We make available some reusable elements of our development publicly (scenarios, transformations, difference models, etc.) [1].

Road-map of the paper

In Sec. 2, we recall the basics of the GMF approach to graphical editor development, and we clarify GMF's use of a collection of coordinated editor models. In Sec. 3, we study a detailed evolution scenario to analyse the co-evolution challenge at hand. In Sec. 4, we develop an initial list of domain model changes and derive a methodology of co-evolution based on propagating changes to all relevant GMF models. In Sec. 5, we describe a principled approach for the automation of the required co-evolution transformation based on the interpretation of difference models for the domain-model changes. In Sec. 6, we sketch a proof-of-concept implementation that is also available online. Related work is described in Sec. 7, and the paper is concluded in Sec. 8.

[1] `http://www.emfmigrate.org/gmfevolution`

2 GMF's coordinated editor models

GMF consists of a generative component (GMF Tooling) and runtime infrastructure (GMF Runtime) for developing graphical editors based on the Eclipse Modeling Framework (EMF) [9, 3] and the Graphical Editing Framework (GEF) [12]. The GMF Tooling supports a model-driven process (see Fig. 2) for generating a fully functional graphical editor based on the GMF Runtime starting from the following models:

- The *domain model* is the Ecore-based metamodel (say, abstract syntax) of the language for which representation and editing have to be provided.
- The *graphical definition model* contains part of the concrete syntax; it identifies graphical elements that may, in fact, be reused for different editors.
- The *tooling definition model* contains another part of the concrete syntax; it contributes to palettes, menus, toolbars, and other periphery of the editor.
- Conceptually, the aforementioned models are reusable; they do not contain references to each other. It is the *mapping model* that establishes all links.

Consider again Fig. 1 which illustrates the role of these models for a simple mind-map editor.[2] Fig. 3 shows all the models involved in the definition and implementation of the mind-map editor. The rectangular boxes highlight contributions that are related to the *Topic* domain concept. It is worth noting how information about domain concepts is scattered over the various models. Most of these recurrences are not remote since most of the correspondences are name-based. Remote references are only to be found in the mapping and the generator models as clarified in the rest of the work.

Domain model This model contains all the concepts and relationships which have to be implemented in the editor. In the example, the class *Mindmap* is introduced as a container of instances of the classes *Topic* and *Relation*.

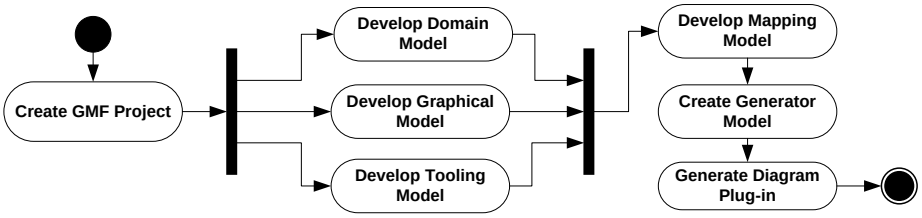

Fig. 2. The model-driven approach to GMF-based editor development.

[2] A mind map is a diagram used to represent words, ideas, tasks, or other items linked to and arranged around a central keyword or idea. The initial mind-map editor suffices with "topics" and "relations", but some extensions will be applied eventually.

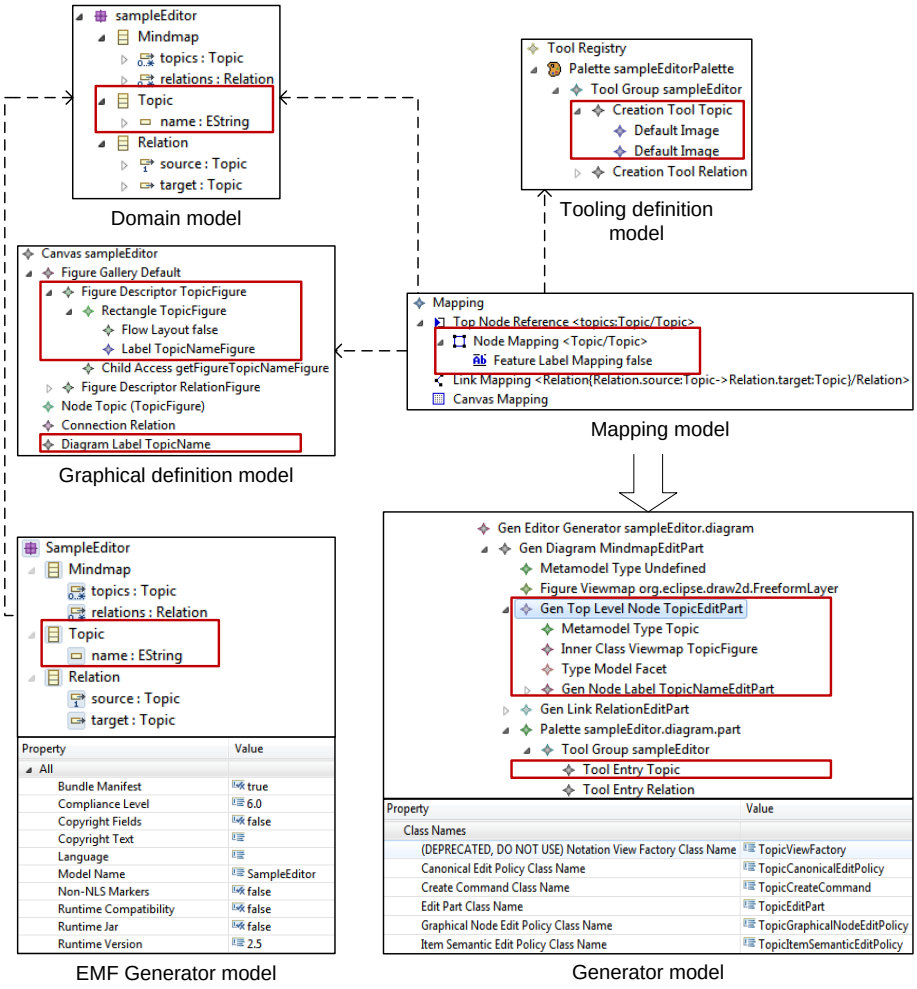

Fig. 3. The GMF models and model dependencies for the editor of Fig. 1.

Graphical definition model This model specifies a figure gallery including shapes, labels, lines, etc., and canvas elements to define nodes, connections, compartments, and diagram labels. For instance, in the graphical model in Fig. 3, a rectangle named *TopicFigure* is defined, and it is referred to by the node *Topic*. A diagram label named *TopicName* is also defined. Such graphical elements will be used to specify the graphical representations for *Topic* instances and their *name*s.

Tooling definition model This model defines toolbars and other periphery to facilitate the management of diagram content. In Fig. 3, the sample model consists of the *Topic* and *Relation* tools for creating the *Topic* and *Relation* elements.

Mapping model This model links together the various models. For instance, according to the mapping model in Fig. 3, *Topic* elements are created by means of the *Creation Tool Topic* and the graphical representation for them is *Node Topic*. For each topic the corresponding *name* is also visualized because of the specified *Feature Label Mapping* which relates the attribute *name* of the class *Topic* with the diagram label *TopicName* defined in the graphical definition model. The meaning of the *false* value near the *Feature Label Mapping* element is that the attribute *name* is not read-only, thus it will be freely edited by the user.

EMF and GMF generator models Once a domain model is defined, it is possible to automatically produce Java code to manage models (instances), say mind maps in our example. To this end an additional model, the *EMF generator model*, is required to control the execution of the EMF generator. A uniform version of the extra model can be generated by EMF tooling.

Once the mapping model is obtained, the GMF Tooling generates (by means of a model-to-model transformation) the *GMF generator model* that is used by a code generator to produce the real code of the modeled graphical editor.

3 GMF's co-evolution challenge

Using a compound change scenario, we will now demonstrate GMF's co-evolution challenges. We will describe how domain-model changes break other editor models, and the editor's code, or make them unsound otherwise. Hence, domain-model changes must be propagated. Such change propagation is not supported currently by GMF, and it is labor-intensive and error-prone, when carried out manually. Conceptually, it turns out to be difficult to precisely predict when and how co-changes must be performed.

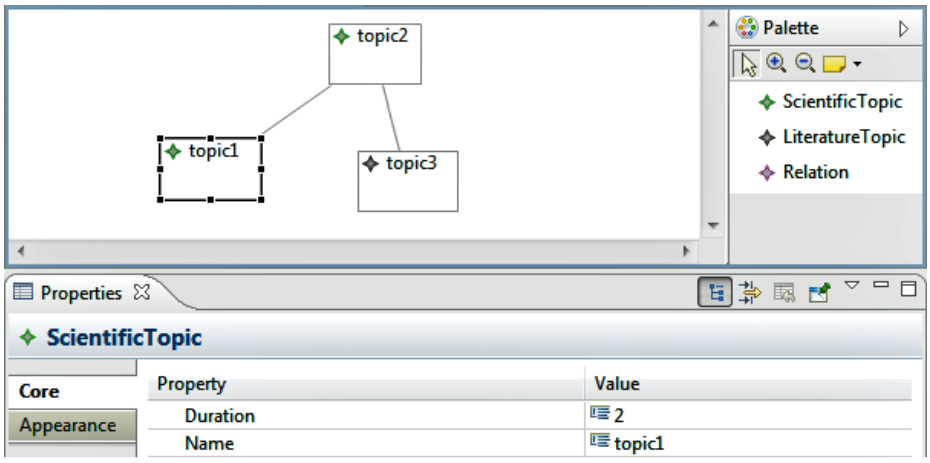

Fig. 4. An evolved mind-map editor with different kinds of topics.

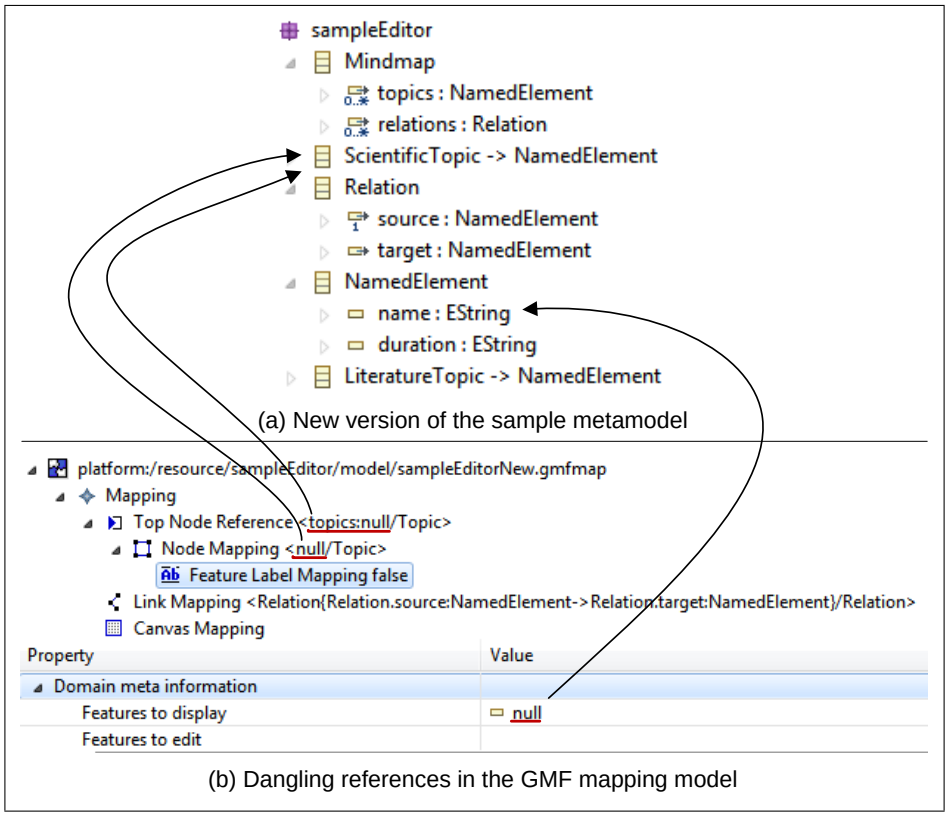

Fig. 5. The domain model for the evolved mind-map editor with the "broken" mapping model.

3.1 A compound change scenario

Consider the enhanced mind-map editor of Fig. 4. Compared to the initial version of Fig. 1, *scientific* vs. *literature topics* are distinguished, and topics have a *duration* property in addition to the *name* property.

Now consider Fig. 5; it shows the evolved metamodel at the top, and the status of the, as yet, unamended mapping model at the bottom. We actually show the mapping model as it would appear to the user if it was inspected in Eclipse. Some of the links in the mapping model are no longer valid; in fact, they are dangling (c.f., "null"). Through extra edges, we show what the links are supposed to be like.

We get a deeper insight into the situation if we comprehend the evolved domain model through a series of simple, potentially atomic changes:

1. The *Topic* class was renamed to *ScientificTopic*.
2. The abstract class *NamedElement* was added.
3. The attribute *name* is pulled up from the *Topic* class to the *NamedElement* class.
4. The attribute *duration* was added to the *NamedElement* class.
5. The class *LiteratureTopic* was added as a further specialization of *NamedElement*.

1	The EMF generator or the GMF generator fails (with an error).
2	The EMF generator or the GMF generator completes with a warning.
3	The generator for the GMF generator model fails.
4	The compiler fails on the generated EMF or GMF code.
5	The editor plugin fails at runtime, e.g., at launch-time.
6	A GMF model editor reports an error upon opening a GMF model.
7	The editor plugin apparently executes, but misses concepts of the domain.
8	The editor plugin apparently executes, but there are GUI events without handlers.

Table 1. Idiosyncratic symptoms of broken and and unsound GMF editors

3.2 Broken vs. unsound GMF models and editors

In practice, these changes would have been carried out in an ad-hoc manner through editing directly the domain model. Because of these changes, the existing mapping model is no longer valid—as shown in Fig. 5. In particular, references to *Topic* or the attribute *name* thereof are dangling. The other GMF models are equally out-of-sync after these domain model changes. For clarity, in Table 1, we sketch a classification of the symptoms that may indicate a broken or unsound GMF editors. Due to space limitation we do not provide an explanation of the reported symptoms which are listed in Table 1 only for the sake of completeness.

Let us consider two specific examples. First, the addition of a new class to the domain model, e.g., *LiteratureTopic*, should probably imply a capability of the editor to create instances of the new class. However, such a creation tool would need to be added in the mapping and tooling models. Second, the renaming of a class, e.g., the renaming of *Topic* to *ScientificTopic*, may lead to an editor with certain functionality not having any effect because elements are referenced that changed or do not exist anymore in the domain model. Both examples are particularly interesting in so far that the editor apparently still works. i.e., it is not *broken* in a straightforward sense. However, we say that the editor is *unsound*; the editor does not meet some obvious expectations.

4 Changes and co-changes

We will now describe a catalogue of domain-model changes and associated co-changes of other editor models. It turns out that there are different options for deciding on the impact of the co-changes. We capture those options by corresponding strategies. As far as the catalogue of changes is concerned, we can obviously depart from catalogues of metamodel changes as they are available in the literature, e.g., [29, 15], and previous work by the authors [4]. For brevity's sake, we make a selection here. That is, we consider only atomic changes that are needed for the compound scenario of the previous section, completed by a few additional ones. Many of the missing changes would refer to technical aspects of the EMF implementation, and as such, they do not contribute to the discussion.

4.1 Strategies for co-changes

Such a distinction of *broken* vs. not broken but nevertheless *unsound* also naturally relates to a spectrum of *strategies* for co-changes. A *minimalistic strategy* would focus

Level	Description
1	Unsound in the sense of being broken; there are reported issues (errors, warnings).
2	Unsound in the sense that the editor "obviously" lacks capabilities.
3	Sound as far as it can be achieved through automated transformations.
4	Sound; established by human evaluation.

Table 2. Levels of editor soundness along evolution.

on fixing the broken editor. That is, co-changes are supposed to bring the editor models to a state where no issues are reported at generation, compile or runtime. A *best-effort strategy* would try to bring the editor to a sound state, or as close to it as possible with a general (perhaps automated) strategy.

Consider again the example of adding a new class *C*:

Minimalistic strategy The execution of the EMF generator emits a warning, which we take to mean that the editor is broken. Hence, we would add the new class *C* to the EMF generator model. This small co-change would be sufficient to re-execute all generators without further errors or warnings, and to build and run the editor successfully. The editor would be agnostic to the new class though because the mapping and tooling models were not co-changed.

Best-effort strategy Let us make further assumptions about the added class *C*. In fact, let us assume that *C* is a concrete class, and it has a superclass *S* with at least one existing concrete subclass *D*. In such a case, we may co-change the other GMF models by providing management for *C* based on the replication of the management for *D*.

Here we assume that a best-effort strategy may be amenable to an automated transformation approach in that it does not require any domain-specific insight. The modeler will still need to perform additional changes to complete the evolution, i.e., to obtain a sound editor.

4.2 Editor soundness related to co-changes

In continuation of the soundness discussion from the previous section, Table 2 identifies different levels of soundness for an evolving editor. The idea here is that we assess the level of the editor *before* and *after* all (automated) co-changes were applied. The proposed transformations can never reach Level 4 because it requires genuine evaluation by the modeler. In other words, Level 4 refers to situations where GMF models can not be automatically migrated and they have to be adapted by the modeler in order to support all the modeling constructs defined in the new version of the considered metamodel.

However, we are not just interested in the overall level of the editor, but we also want to *blame* one or more editor models for the editor's unsoundness. In Table 3, we list atomic changes with the soundness levels for the editor before and after co-changes, and all the indications as to what models are to blame. We use "×" to blame a model for causing the editor to be broken, i.e., to be at Level 1. We use "○" and "●" likewise for Level 2 and Level 3.

	before co-change					after co-change				
	EMFGen	Graph	Tooling	Mapping	Level	EMFGen	Graph	Tooling	Mapping	Level
Add empty, concrete class	×	○	○	○	1	●	○	○	○	2
Add empty, abstract class	×	●	●	●	1	●	●	●	●	3
Add specialization	●	●	●	●	3	●	●	●	●	3
Delete concrete class	×	○	×	×	1	●	○	●	●	2
Rename class	×	○	○	×	1	●	●	●	●	3
Add property	×	○	○	○	1	●	○	○	●	2
Delete property	×	○	×	×	1	●	○	●	●	2
Rename property	×	○	○	×	1	●	●	●	●	3
Move property	×	○	×	×	1	●	○	○	○	2
Pull up property	×	○	×	×	1	●	○	●	●	2
Change property type	●	○	×	×	1	●	○	○	○	2

Table 3. Considered Ecore metamodel changes

The EMFGen model is frequently to blame for a broken editor *before* the co-changes; the Graph model is never to blame for a broken editor; the remaining models are to blame occasionally for a broken editor. Obviously, there is trend towards less blame after the co-changes: no occurrences of "×", more occurrences of "●". In different terms, for all domain-model changes, all other models can be co-changed so that the editor is no longer broken. In several cases, we reach Level 3 for the editor.

There are clearly constellations for which changes cannot be propagated in an automated manner that resolves all Level 2 blame. For instance, the metamodel change *add empty, concrete class* does not require a co-changed Graph model as long as some existing graphical element can be reused. However, avoidance of Level 2 blame would require a manual designation of a new element or genuine selection of a specific element as opposed to an automatically chosen element.

4.3 Specific couples of changes and co-changes

In the rest of the section, the changes reported in Table 3 and the corresponding co-changes, which have to be operated on the GMF models, are described in more detail.

Add empty, concrete class Apart from the EMFGen model, the other ones are not affected; the editor simply does not take into account the added class. Thus, modelers cannot create or edit instances of the new class. The co-change may replicate the model from existing classes as discussed in Sec. 3. Ultimately, the modeler may need to manually complete the management of the new class.

Add empty, abstract class In comparison to the previous case, the co-change of the EMFGen model is fully sufficient since abstract classes cannot be instantiated, and hence, no additional functionality is needed in the editor.

Add specialization The change consists of modifying an existing class by specifying it as specialization of another one. In particular, in the simple case of the superclass being empty, this modification does not affect any model; thus, no co-changes are required.

Delete concrete class Deleting an existing class is more problematic since all the GMF models except the Graph model are affected. Especially the Mapping model has to be fixed to solve possible dangling references to the deleted class. The Tooling model is also co-changed by removing the creation tool used to create instances of the deleted class. Even if the model is not adapted, the generator model and thus the editor can be generated—even though the palette will contain a useless tool without associated functionality. The Graph model can be left unchanged. The graphical elements which were used for representing instances of the deleted class, may be re-used in the future.

Rename class Renaming a class requires co-change of the Mapping model which can have, as in the case of class deletion, invalid references which have to be fixed by considering the new class name. The Graph model does not require any co-change since the graphical elements used for the old class can be used even after the rename operation. The Tooling model can be left untouched, or alternatively the label and the description of the tool related to the renamed class can be modified to reflect the same name. However, even with the same Tooling model, a working editor will be generated.

Add property The strategy for co-change is similar to the addition of new classes.

Delete property Deleting a property which has a diagrammatic representation requires a co-change of the Mapping model in order to fix occurred dangling references. Moreover if some tools were available to manage the deleted property, also the Tooling model has to be co-changed. As in the case of class removals, the graphical model can be left unchanged.

Rename property The strategy for co-change is similar to the renaming of classes.

Move property When a property is moved from one class to another, then dangling references may need to be resolved in the Mapping model. If the moved property is managed by means of some tools, the Tooling model require co-changes, too. We only offer a simple, generic strategy for co-changes: the repaired editor does not consider the moved property.

Pull up property Given a class hierarchy, a given property is moved from an extended to a base class. This modification is similar to the previous one—even though an automatic resolution can be provided to co-change Tooling and Mapping models in a satisfactory manner.

Change property type The EMFGen model is not affected. However, by changing the type of a property some dangling references can occur in the Mapping model; their resolution cannot be fully automated. Also, if the affected property is managed by some tool, then the Tooling model must be co-changed as well.

5 Automated adaptation of GMF models

Having a catalogue of changes like the one previously discussed is preparatory for supporting the adaptation of GMF models. In particular, it can be exploited to automatically

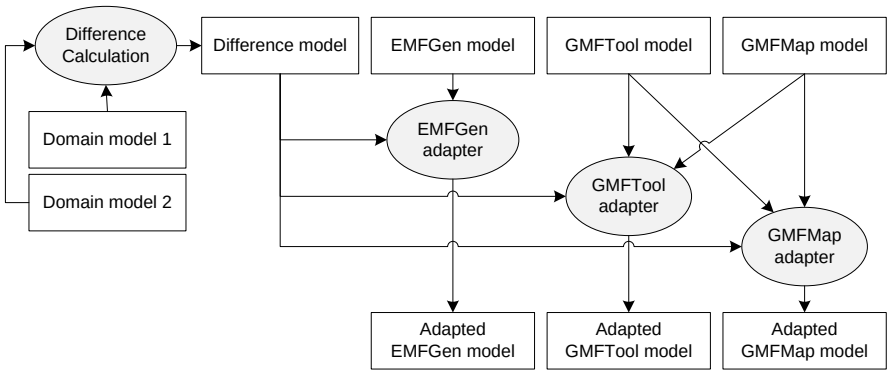

Fig. 6. Overview of the process of co-evolution with automated transformations.

detect the modifications that have been operated on a given domain model, and to instruct corresponding migration procedures as proposed in the rest of the paper.

We have developed a general process for GMF co-evolution which involves model differencing techniques and automated transformations to adapt existing GMF models with respect to the changes operated on domain models. The approach is described in Fig. 6 and consists of the following elements:

- *Difference calculation*, given two versions of the same *domain model*, their differences are calculated to identify the changes which have been operated on the first version of the model to obtain the last one. The calculation can be operated by any of the existing approaches able to detect the differences between any kind of models, like EMFCompare [7];

- *Difference representation*, the detected differences have to be represented in a way which is amenable to automatic manipulations and analysis. To take advantage of standard model driven technologies, the calculated differences should be represented by means of another model;

- *Generation of the adapted GMF models*, the differences represented in the difference model are taken as input by specific adapters each devoted to the adaptation of a given GMF model with respect to the metamodel modifications and corresponding co-changes reported in Table 3. In particular, the GMFMap and the GMFTool adapters are devoted to the adaptation of the *GMFMap model* and the *GMFTool model*, respectively. Such adapters take both models because of dependencies between them which have to be updated simultaneously. The *EMFGen model* is updated by means of a specific adapters, whereas no adapter is provided for the *Graph model*. In fact, the discussion of the previous sections suggested that we can always reasonable continue with the old *Graph model*. The adapters can be implemented as model transformations which take as input the old version of the GMF models and produce the adapted ones.

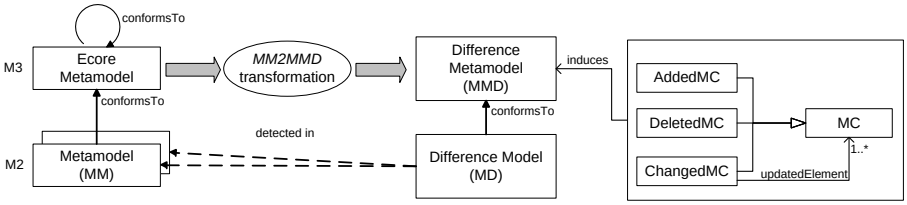

Fig. 7. Difference metamodel generation

Interestingly, the process in Fig. 6 is independent from the technologies which have been adopted both for calculating and managing domain model differences, and to automatically manipulate them for generating the adapted GMF models.

6 Proof-of-concept implementation of the GMF adapters

In this section we propose the support for the GMF model adaptation approach we described in the previous section. That is, in Sec. 6.1, we outline a technique for representing the differences between two versions of a same metamodel. Such a representation approach has been already used by the authors for managing other co-evolution problems [4]. Further, in Sec. 6.2, the ATL transformation language [17] is adopted for implementing the different model adapters which have been identified to evolve existing GMF models. The implementation of the approach is available publicly as described in the introduction of the paper.

6.1 Model-based representation of domain model differences

The differences between different versions of a same domain model can be represented by exploiting the *difference metamodel* concept, presented by the authors in [5]. The approach is summarized in Fig. 7: given two Ecore metamodels, their difference conforms to a difference metamodel *MMD* derived from Ecore by means of the *MM2MMD* transformation. For each class *MC* of the Ecore metamodel, the additional classes *AddedMC*, *DeletedMC*, and *ChangedMC* are generated in the extended Ecore metamodel by enabling the representation of the possible modifications that can occur on domain models and that can be grouped as follows:

- *additions*, new elements are added in the initial metamodel;
- *deletions*, some of the existing elements are deleted;
- *changes*, some of the existing elements are updated.

In Fig. 8, a fragment of the difference model representing the changes between the domain models in Fig. 3 and Fig. 5 is shown. Such a difference model conforms to a difference metamodel automatically obtained from the ECore metamodel. For instance, from the metaclass *EClass* of the ECore metamodel, the metaclasses *AddedEClass*, *DeletedEClass*, and *ChangedEClass* are generated in the corresponding difference metamodel.

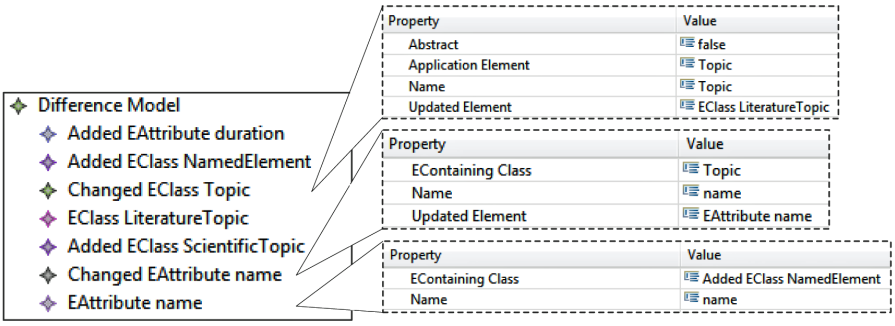

Fig. 8. Fragment of the difference model for the evolution scenario of Sec. 3.

For some of the reported differences in Fig. 8, the corresponding properties are shown. For instance, the renaming of the *Topic* class is represented by means of a *ChangedEClass* instance which has as updated element an instance of *EClass* named *LiteratureTopic* (see the *updatedElement* property of the changed class *Topic* shown on the right-hand side of Fig. 8). The addition of the class *NamedElement* is represented by means of an *AddedEClass* instance. The move operation of the attribute *name* from the class *Topic* to the added class *NamedElement* is represented by means of a *ChangedEAttribute* instance which has one *EAttribute* instance as updated element with a different value for the *eContainingClass* property. In fact, in the initial version it was *Topic* (see the second property window) whereas in the last one, it is *NamedElement* (as specified in the third property window).

6.2 ATL-based implementation of GMF model adapters

Our prototypical implementation of the GMF model adapters leverages ATL [17], a QVT [24] compliant language which contains a mixture of declarative and imperative constructs. In particular, each model adapter is implemented in terms of model transformations which use a common query library described in rest of the section.

An ATL transformation consists of a module specification containing a header section (e.g. lines 1-3 in Listing 1.1), transformation rules (lines 5-42 in Listing 1.1) and a number of helpers which are used to navigate models and to define complex calculations on them (some helpers which have been implemented are described in Table 4). In particular, the header specifies the source models, the corresponding metamodels, and the target ones; the helpers and the rules are the constructs used to specify the transformation behaviour.

Small excerpts of the GMFMap and GMFTool adapters are shown in Listing 1.1 and Listing 1.2, respectively. For instance, the *AddedSpecializationClassTo...* transformation rules manage new classes which have been added in the domain model as specializations of an existing one. The code excerpts involve the replication strategy that we have described in previous sections.

ATL transformation rules consist of source and target patterns: the former consist of source types and an OCL [25] guard stating the elements to be matched; the latter are

Helper name	Context	Return type	Description
getEClassInNewMetamodel	EClass	EClass	Given a class of the old metamodel, it returns the corresponding one in the new metamodel.
getNewContainer	EAttribute	EClass	Given an EAttribute in the old metamodel, the corresponding container in the new one is retrieved. To this end, the helper checks if the EAttribute has been moved to a new added class, if not an existing class is returned.
isMoved	EAttribute	Boolean	It checks if the considered EAttribute has been moved to another container
isMovedToAddedEClass	EAttribute	Boolean	It checks if the considered EAttribute has been moved to a new added EClass.
isRenamed	EAttribute	Boolean	It checks if the given EAttribute has been renamed.

Table 4. Some helpers of the *gmfAdaptationLib*

composed of a set of elements, each of them specifies a target type from the target metamodel and a set of bindings. A binding refers to a feature of the type, i.e. an attribute, a reference or an association end, and specifies an expression whose value initializes the feature. For instance, the *AddedSpecializationClassToNodeMapping* rule in Listing 1.1 is executed for each match of the source pattern in lines 8-17 which describes situations like the one we had in the sample scenario where the *LiteratureTopic* class (see *s1*) is added as specialization of an abstract class (see *s2*) which is specialized by another class (see *s3*). In this case, the Mapping model is updated by adding a new *TopNodeReference* and its contained elements (see lines 24-41) which are copies of those already existing for *s3*.

A similar source pattern is used in the rule of Listing 1.2 (lines 7-12) in order to add a creation tool for the new added class *s1* to the Tooling model (see lines 19-23).

```
1 module GMFMapAdapter;
2 create OUT : GMFMAPMM from IN : GMFMAPMM, GMFTOOL: GMFTOOLMM, DELTA: DELTAMM,
3       NEWECORE : ECORE, OLDECORE : ECORE ;
4 ...
5 rule AddedSpecializationClassToNodeMapping {
6
7   from
8    s1: DELTAMM!AddedEClass, s2: DELTAMM!AddedEClass,
9    s3: DELTAMM!ChangedEClass, s4: DELTAMM!ChangedEAttribute,
10   s5: DELTAMM!EAttribute
11     ((not s1.abstract)
12      and s1.eSuperTypes->first() = s2
13      and s2.abstract
14      and s3.updatedElement->first().eSuperTypes->first() = s2
15      and s4.updatedElement->first() = s5
16      and s4.eContainingClass = s3
17      and s5.eContainingClass = s2 ))
18
19  using {
20    siblingFeatureLabelMapping : GMFMAPMM!FeatureLabelMapping =
```

```
            s3.getNodeMappingFromChangedClass().labelMappings
             ->select(e | e.oclIsTypeOf(GMFMAPMM!FeatureLabelMapping))->first(); }

  to
   t1 : GMFMAPMM!TopNodeReference (
      containmentFeature <- s3.getTopNodeReferenceFromChangedClass().
                containmentFeature.getFeatureInNewMetamodel(),
      ownedChild <- t2
    ),
   t2 : GMFMAPMM!NodeMapping (
      domainMetaElement <- s1.getAddedClassInNewMetamodel(),
      relatedDiagrams <- s3.getNodeMappingFromChangedClass().relatedDiagrams,
      tool <- s1.name.getNewToolFromTitle(),
      diagramNode <- s3.getNodeMappingFromChangedClass().diagramNode
    ),
   t3 : GMFMAPMM!FeatureLabelMapping (
      diagramLabel <- siblingFeatureLabelMapping.diagramLabel,
      features <- siblingFeatureLabelMapping.features->collect(e |
            e.getFeatureInNewMetamodel())
    ),
    ...
}
```

Listing 1.1. Fragment of the GMFMap Adapter

To summarize, the implementation of the GMF adapters consists of transformation rules which copy the given source model to a target one; during this operation they evaluate if changes are needed. A number of helpers have been defined; they navigate models and perform complex queries on them. Many of the helpers are common to all the adapters, and hence, they are available through a library *gmfAdaptationLib*. Table 4 describes some of these helpers.

```
module GMFToolAdapter;
create OUT : GMFTOOLMM from IN : GMFTOOLMM, GMFMAP : GMFMAPMM, DELTA: DELTAMM,
      NEWECORE : ECORE, OLDECORE : ECORE ;
...
rule AddedSpecializationClassToCreationTool {

  from
   s1: DELTAMM!AddedEClass, s2: DELTAMM!AddedEClass, s3: DELTAMM!ChangedEClass
    ( (not s1.abstract)
      and s1.eSuperTypes->first() = s2
      and s2.abstract
      and s3.updatedElement->first().eSuperTypes->first() = s2 )

  using {
   toolGroup : GMFTOOLMM!ToolGroup = OclUndefined;
  }

  to
   t : GMFTOOLMM!CreationTool (
     title <- s3.getToolFromChangedClass().title.regexReplaceAll(s3.
                getToolFromChangedClass().title, s1.name),
     description <- 'Create␣new␣' + s1.name
     ),
     ...
}
```

Listing 1.2. Sample transformation rule of the GMFTool Adapter

7 Related work

7.1 Graphical model editors

In [1], a number of technologies for the development of domain-specific modeling languages (DSMLs) are evaluated; Eclipse (EMF with GEF) is covered, but not GMF. The evaluation criteria include language evolution to mean the ability to co-evolve models when the domain model changes. There is no criterion though that relates to GMF's particular characteristics of using multiple editor models.

Other GMF- or GEF-based frameworks have been proposed. For instance, the MuvitorKit framework [23] is based on EMF and GEF and specifically meant as an alternative to GMF for the benefit of additional editor capabilities (e.g., multiple panes) as well as additional modeling capabilities, thereby requiring less customization of generated code. There is also the EuGENia framework [20, 19] which raises the level of abstraction in GMF-based development by using annotations on the domain model, thereby feeding into code generation. We are not aware of any prior effort to propagate changes across GMF models.

The ViatraDSM framework [27] replaces GMF in that it allows for versatile mappings between abstract and concrete syntax. Live transformations are leveraged to maintain the coherence of the two models. Our uni-directional, difference-driven transformations propagate domain-model changes elsewhere. Our work is specifically targeted at the mainstream GMF-based approach with its various models.

7.2 Model consistency

The status of GMF models being out-of-sync can be compared to the notion of model inconsistency in (UML-based) modeling where different models providing different views may require synchronization. For instance, in [8], inconsistencies between the different diagrammatic forms in UML models are considered, and possible fixes are proposed in the form of value changes. In [13], the dependencies between models are modeled through triple graph grammars in a manner that enables incremental model synchronization. Our specific contribution is one of reverse engineering: discovering the GMF model dependencies, and making them operational through automated transformations.

7.3 Co-evolution of metamodels and models

The techniques and the methodology of our work are inspired by research on co-evolution in model-driven engineering [10, 28]. Much of this work is concerned with co-transforming models in reply to metamodel changes [29, 15]. In that case metamodel changes can invalid existing models that have to be adapted to recover the conformance with the new version of the metamodel.

In this work, we analyze another kind of co-evolution, even though related to the previous one, which aims at propagating metamodel changes to the other GMF models according to a given soundness level of the editor. The overall proposal leverages the difference representation approach proposed by the authors in [5] and already used to manage co-evolution problems in [6].

7.4 Syntax relationships for textual languages

In [18, 30, 26], approaches for the operationalization of the link between concrete and abstract syntax definition are described. That is, concrete syntax definitions are customized into abstract syntax definitions. In fact, the approach of [18] is based on the idea that concrete and abstract syntax definitions can be incomplete but they automatically complete each other based on name mapping and other heuristics. In contrast, the approach of [30] is based on grammar transformations where the concrete syntax is mapped operationally to the abstract syntax. In [26], yet another approach is exercised, where the abstract syntax definition is associated with the concrete syntax definition through annotations. (A similar MDE approach is the one of TCS for KM3 [16].) None of these approaches provides any automated capabilities for change propagation. The classical approach to concrete-to-abstract syntax mappings is to use an attribute grammar. There are a number of approaches to align grammar transformations with attribution transformations, see, e.g., [21, 22], but none of these approaches are directly applicable to the synchronization of abstract and concrete syntax. We contend that the problem of collections of coordinated GMF editor models seems to be even more complicated than concrete/abstract syntax synchronization.

8 Concluding remarks

We have described the challenge of sound evolution for graphical editors based on model-driven development with GMF in particular, and we have addressed this challenge by a system of co-transformations that propagate changes from domain models to the other editor models.

We have identified a range of options for evolved editors to be unsound, and we have described corresponding resolution strategies. In the more established area of metamodel/model co-evolution, models either are not broken, or they are broken and can be reasonably resolved in an automated manner, or a well-understood problem-specific contribution to the resolution must be provided manually or through a heuristic. In the case of co-evolution for editor models, there is a scale of models being broken or unsound. Also, each of the various models calls for a designated analysis. Finally, there are intricate inter-model dependencies.

The existing GMF infrastructure is obviously rather complicated: it consists of a number of metamodels, libraries, generators, model transformations of industrial scale. We cannot claim to provide a full-fledged solution to the co-evolution challenge of GMF—this would require full coverage of Ecore, the metamodeling language of EMF, and full understanding of the implicit semantics of GMF model dependencies and tools.

The focus of this paper is on the conceptual co-evolution challenge at hand. The development of industrial-strength tools for co-evolution or the revision of the GMF suite is a clearly a major undertaking that is beyond the scope of this paper. The prototypical implementation of the proposed approach supports all the metamodel changes reported in Table 3. Nevertheless, we are confident that our transformational approach can be scaled incrementally over time to cover an increasing number of concrete evolution scenarios. In the future we plan to support them by providing additional effort in the

implementation of the overall approach. The most critical omission in our methodology is that we do not currently cover co-evolution of custom code. This is a very intricate problem by itself, to which we hope to contribute through future work.

In our ongoing research, we try to better understand the co-evolution issues and associated strategies for the *code level* of GMF where generated code has been possibly customized. Based on preliminary research, we can already report that customization is used by some GMF projects extensively, and hence designated co-evolution support may provide significant help with real-world editor development.

References

1. D. Amyot, H. Farah, and J.-F. Roy. Evaluation of Development Tools for Domain-Specific Modeling Languages. In *System Analysis and Modeling: Language Profiles, 5th International Workshop, SAM 2006, Revised Selected Papers*, volume 4320 of *LNCS*, pages 183–197. Springer, 2006.
2. J. Bézivin. On the Unification Power of Models. *Jour. on Software and Systems Modeling (SoSyM)*, 4(2):171–188, 2005.
3. F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. Grose. *Eclipse Modeling Framework*. Addison Wesley, 2003.
4. A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Automating Co-evolution in Model-Driven Engineering. In *12th International IEEE Enterprise Distributed Object Computing Conference, ECOC 2008, Proceedings*, pages 222–231. IEEE Computer Society, 2008.
5. A. Cicchetti, D. Di Ruscio, and A. Pierantonio. A Metamodel Independent Approach to Difference Representation. *Journal of Object Technology*, 6(9):165–185, October 2007.
6. A. Cicchetti, D. D. Ruscio, and A. Pierantonio. Model Patches in Model-Driven Engineering. In *Models in Software Engineering, Workshops and Symposia at MODELS 2009, Reports and Revised Selected Papers*, volume 6002 of *LNCS*, pages 190–204. Springer, 2010.
7. Eclipse Foundation. EMF Compare, 2010. `http://www.eclipse.org/modeling/emft/?project=compare`.
8. A. Egyed, E. Letier, and A. Finkelstein. Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, pages 99–108. IEEE, 2008.
9. Eclipse project: Eclipse Modeling Framework Project (EMF). `http://www.eclipse.org/modeling/emf/`.
10. J.-M. Favre. Meta-Model and Model Co-evolution within the 3D Software Space. In *Proceedings of International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA 2003), Co-located with the IEEE International Conference on Software Maintenance (ICSM 2003)*, 2003.
11. J.-M. Favre. Languages Evolve Too! Changing the Software Time Scale. In IEEE, editor, *8th Interntational Workshop on Principles of Software Evolution, IWPSE*, 2005.
12. Eclipse project: GEF - Graphical Editing Framework. `http://www.eclipse.org/gef/`.
13. H. Giese and R. Wagner. Incremental Model Synchronization with Triple Graph Grammars. In *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Proceedings*, volume 4199 of *LNCS*, pages 543–557. Springer, 2006.
14. Eclipse project: GMF - Graphical Modeling Framework. `http://www.eclipse.org/gmf/`.

15. M. Herrmannsdoerfer, S. Benz, and E. Jürgens. COPE - Automating Coupled Evolution of Metamodels and Models. In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Proceedings*, volume 5653 of *LNCS*, pages 52–76. Springer, 2009.
16. F. Jouault, J. Bézivin, and I. Kurtev. TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In *Proceedings of Generative programming and component engineering (GPCE 2006)*, pages 249–254. ACM, 2006.
17. F. Jouault and I. Kurtev. Transforming Models with ATL. In J.-M. Bruel, editor, *MoDELS Satellite Events*, volume 3844 of *LNCS*, pages 128–138. Springer-Verlag, 2005.
18. B. Kadhim and W. Waite. Maptool—supporting modular syntax development. In T. Gyimothy, editor, *Proceedings, Compiler Construction (CC'96)*, volume 1060 of *LNCS*, pages 268–280. Springer, Apr. 1996.
19. D. S. Kolovos, L. M. Rose, S. B. Abid, R. F. Paige, and G. Botterweck. Taming EMF and GMF Using Model Transformation. In *13th ACM/IEEE International Conference on Model Driven Engineering, Languages and Systems (MoDELS)*, 2010. to appear.
20. D. S. Kolovos, L. M. Rose, R. F. Paige, and F. A. C. Polack. Raising the level of abstraction in the development of GMF-based graphical model editors. In *MISE '09: Proceedings of the 2009 ICSE Workshop on Modeling in Software Engineering*, pages 13–19. IEEE, 2009.
21. R. Lämmel and G. Riedewald. Reconstruction of paradigm shifts. In *Proceedings of the Second Workshop on Attribute Grammars and their Applications (WAGA 1999)*, pages 37–56, Mar. 1999. INRIA Technical Report ISBN 2-7261-1138-6.
22. W. Lohmann, G. Riedewald, and M. Stoy. Semantics-preserving migration of semantic rules after left recursion removal in attribute grammars. In *Proceedings of 4th Workshop on Language Descriptions, Tools and Applications (LDTA 2004)*, volume 110 of *ENTCS*, pages 133–148. Elsevier Science, 2004.
23. T. Modica, E. Biermann, and C. Ermel. An Eclipse Framework for Rapid Development of Rich-featured GEF Editors based on EMF Models. In *Informatik 2009: Im Focus das Leben, Beiträge der 39. Jahrestagung der Gesellschaft füur Informatik e.V. (GI), Proceedings*, volume 154 of *LNI*, pages 2972–2985. GI, 2009.
24. Object Management Group (OMG). MOF QVT Final Adopted Specification, 2005. OMG Adopted Specification ptc/05-11-01.
25. Object Management Group (OMG). OCL 2.0 Specification, 2006. OMG Document formal/2006-05-01.
26. J. L. Overbey and R. E. Johnson. Generating Rewritable Abstract Syntax Trees. In *Software Language Engineering, First International Conference, SLE 2008, Revised Selected Papers*, volume 5452 of *LNCS*, pages 114–133. Springer, 2009.
27. I. Ráth, A. Ökrös, and D. Varró. Synchronization of abstract and concrete syntax in domain-specific modeling languages—By mapping models and live transformations. *Journal of Software and Systems Modeling*, 2009.
28. S. Vermolen and E. Visser. Heterogeneous Coupled Evolution of Software Languages. In *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Proceedings*, volume 5301 of *LNCS*, pages 630–644. Springer, 2008.
29. G. Wachsmuth. Metamodel Adaptation and Model Co-adaptation. In *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Proceedings*, volume 4609 of *LNCS*, pages 600–624. Springer, 2007.
30. D. Wile. Abstract syntax from concrete syntax. In *Proceedings, International Conference on Software Engineering (ICSE'97)*, pages 472–480. ACM Press, 1997.