Dealing with the coupled evolution of metamodels and model-to-text transformations

Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio

DISIM University of L'Aquila,
{name.lastname}@univaq.it

Abstract. In Model-Driven Engineering (MDE) the modification of a metamodel typically can invalidate many different sorts of artifacts. In order to mitigate the pragmatic consequences of such problem, several coupled evolution techniques have been introduced over the last few years mainly focussing on restoring the validity of models, transformations, and editors. However, none of the proposed techniques addressed the coupled evolution of metamodels and template-based code generators, which has been largely neglected despite the relevance of such systems. In an attempt to fill the gap, this paper presents an approach for the coupled evolution of Acceleo-based templating including the OCL embedded in its notation. The approach has been implemented and illustrated by means of a running example.

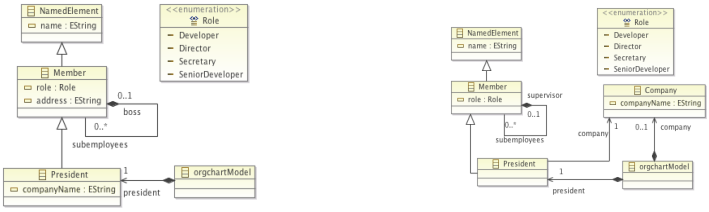
1 Introduction

In Model-Driven Engineering [1] (MDE) the employment of metamodels is pervasive. They are used to formally describe a wide range of artifacts, including models, transformations, concrete syntaxes, and editors. In essence, metamodels are at the core of any modeling ecosystem and their management is therefore key to success. Similarly to any software component metamodels are expected to evolve during their lifecycle [2]. However, because of the dependencies between metamodels and the related artifacts, modifying a metamodel might compromise the validity of the latters. Unfortunately, restoring the validity of such artifacts in a semi-automated manner is intrinsically difficult because it must consider both the purpose of the metamodel modification (i.e., updative, adaptive, performance, corrective or reductive) and the technical aspects (i.e., the when, where, what and how of changes) [3]. While several approaches already investigated the coupled evolution of models (e.g., [4,5,6]), transformations (e.g., [7,8,9]), and editors (e.g., [7,10]), the coupled evolution of template-based code generators has been largely neglected at the expense of reduced pragmatics.

In this paper, we propose an approach to the coupled evolution of metamodels and template-based code generators. In particular, we provide a solution to the problem of adapting Acceleo[1]-based templates when the metamodels of the input models are changed. The proposed approach is able to adapt corrupted Acceleo templates and it is performed by means of an ATL *adaptor*, i.e., a model transformation which takes the metamodel changes (represented in an opportune model-based notation), the model

[1] http://www.obeo.fr/pages/acceleo

II



(a) Initial version

(b) Evolved version

Fig. 1: OrgChart simple metamodel evolution

associated to the corrupted Acceleo template and returns the adapted template. The approach presents many similarities to other approaches focussing on the adaptation of other kinds of artifacts, however it presents also specific difficulties: a) all the possible metamodel refactorings (see [11]) must be dealt by the adaptor, and b) Acceleo provide model traversing facilities by means of the Object Constraint Language[2] (OCL) whose expressions must be adapted as well. Interestingly, to the best of our knowledge none of the current approaches focussed on code-generators, despite the relevance they have in a wide range of projects.

The paper is organized as follows. In Section 2 we clarify the context of this work, exposing briefly Acceleo and the co-evolution problem using a real case study. In Section 3 we propose the process for the resolution of the presented problem and we show the models before and after the resolution. In Section 4 a related section is organized for the comparison of the existing works and the presented one. We conclude also including the future works in Section 5.

2 The coupled evolution problem in model-to-text transformations

Almost any artifact involved in a model-driven development processes at different extent depends on the considered metamodels. Dependencies can emerge at different times during the metamodel life-cycle, and with different degrees of causality depending on the nature of the considered artifact. An ecosystem of modeling artifacts and tools developed atop of a metamodel, like the OrgChart *metamodel* shown in Fig. 1.a and presented later in the paper, is a living set of artifacts working together and built on the top the considered domain. For example *graphical* and *textual editors* might be developed to support the specification of OrgChart *models*. *Model transformations* might be also developed to generate target models or code out of source OrgChart models. Moreover, the Acceleo-based *WebPages code generator* is one of the possible code generator used to generate target Web pages from source OrgChart models.

When metamodels are changed e.g., to satisfy unforeseen requirements or simply to better represent concepts of the considered domain, the already existing artifacts might be compromised and they have to be adapted in order to recover their conformance with the new version of the changed metamodel. The metamodel evolution problem and the consequent ecosystem migration has been discussed [12] and to the best of our knowledge the adaptation of Acceleo-based generators has not been investigated yet and it represents the main goal of this paper.

[2] `http://www.omg.org/spec/OCL/`

```
1  [comment encoding = UTF-8 /]
2  [module generate('http://www.disim.univaq.it/orgchart')]
3  [template public generateElement(aPresident : President)]
4  [comment @main/]
5  [file (aPresident.companyName.concat('.html'), false, 'UTF-8')]
6  <html>
7    <head>
8      <script type='text/javascript' src='https://www.google.com/jsapi'></script>
9      <script type='text/javascript'>
10       google.load('visualization', '1', {packages: [ '[' /]'orgchart' [ ']' /]});
11       google.setOnLoadCallback(drawChart);
12       function drawChart() {
13         var data = new google.visualization.DataTable();
14         data.addColumn('string', 'Name');
15         data.addColumn('string', 'Manager');
16         data.addColumn('string', 'ToolTip');
17         data.addRows( [ '[' /]
18         [ '[' /]{v:'[aPresident.name/]', f:'[aPresident.name/]<div style="color:red; font-style:italic">President</div>'}, '', 'The
19           [for (subemp : Member | aPresident.subemployees) separator (',')]
20           [generateSubMembers(subemp)/]
21           [/for]
22         [ ']' /]
23         );
24         var chart = new google.visualization.OrgChart(document.getElementById('chart_div'));
25         chart.draw(data, {allowHtml:true});
26       }
27     </script>
28   </head>
29   <body><center><h2>[aPresident.companyName/] Orgchart</h2></center><div id='chart_div'></div></body>
30 </html>
31 [/file]
32 [/template]
33 [template public generateSubMembers(m : Member)]
34 [if (m.subemployees->size()>0)]
35 [ '[' /]{v:'[m.name/]', f:'[m.name/]<div style="color:red; font-style:italic">[m.role.toString()/]</div>'}, '[m.boss.name/]',
36 '[m.address/]'[ ']' /],
37    [for (subm : Member | m.subemployees) separator (',')]
38       [generateSubMembers(subm)/]
39    [/for]
40 [else]
41  [ '[' /]{v:'[m.name/]', f:'[m.name/]<div style="color:red; font-style:italic">[m.role.toString()/]</div>'}, '[m.boss.name/]',
42  '[m.address/]'[ ']' /]
43 [/if]
44 [/template]
```

Fig. 2: Sample Acceleo templates

The remaining of the section is organized as follows: an overview of Acceleo is given in Section 2.1. Section 2.2 introduces the problems related to the adaptation of Acceleo-based transformations according to the changes operated on the corresponding metamodels.

2.1 Acceleo-based model-to-text transformations

Acceleo is a model-to-text transformation tool typically used to develop code generators. Acceleo generators are based on templates that identify repetitive and static parts of the applications, and embody specific queries on the source models to fill the dynamic parts. Fig. 2 shows a fragment of the Acceleo template that has been developed to generate graphical representations of source OrgChart models as shown in Fig. 3. In particular, the template generates HTML and Javascript code that uses the Google Chart API[3] to get the chart representations like the one in Fig. 3.b from source organizational models. Each Acceleo-based text generator starts with the specification of a *module* referring to the metamodel that will be used during the generation process (see line 2 in Fig. 2). Lines 3-32 consist of the *template* used to transform instances of the *President* metaclass in Fig. 3a. In particular, for each president in the source model a corresponding HTML file is generated (see line 5). The file name is obtained from the attribute *companyName* in the *aPresident* variable, that is an instance of *President*.

From line 8 to 17 the template consists of static JavaScript code to create the chart. Interestingly, lines 18 to 21 consist of an iteration that creates the rows related to the

[3] `https://developers.google.com/chart/`

members to be represented as boxes in the charts. Each row contains information taken from the *Member* instance e.g., the *name* and the *role*. Once the president element has been transformed, the other members are generated by means of the template *generateSubMembers* (see lines 33-44)specifically developed to manage instances of the metaclass *Member*. Such a template iterates on the *subemployees* relations in order to generate corresponding boxes.

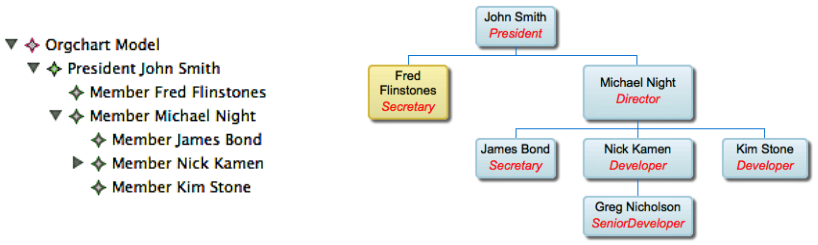
2.2 Invalidating Acceleo-based templates with metamodel evolution

As previously mentioned, existing modeling artifacts might be affected by the changes operated on the corresponding metamodels, and Acceleo templates are not an option. For instance, the metamodel evolution shown in Fig. 1b compromises the Acceleo templates previously analyzed that become invalid in different points. In particular, the new version of the OrgChart metamodel has been obtained by operating the following metamodel change patterns [6]:

i) a new metaclass with name *Company* has been added with the attribute *companyName* moved into it. Such a modification refers to the *extract metaclass* pattern;
ii) the attribute *address* in the initial *Member* metaclass has been removed by applying the *remove attribute* pattern;
iii) the reference *boss* has been renamed as *supervisor* by applying the *rename reference* pattern.

Because of such changes the templates presented in the previous section are invalid and cannot be applied on models conforming to the new version of the OrgChart metamodel. In particular, the references to the *companyName* attribute (e.g., see the expression *"aPresident.companyName"* at line 5 in Fig. 2) have to be migrated since the attribute has been moved to a new metaclass. A similar problem occurs at lines 36-42 that refers to the attribute *address* that has been removed. Finally, lines 35-41 have to be also migrated since they refer to the feature *boss* that has been renamed as *supervisor*.

Adapting Acceleo templates without a proper supporting can be a strenuous and error-prone task. In the next section we propose an approach based on model-differencing and model transformations that under certain conditions is able to automatically adapt affected Acceleo templates.



(a) Source OrgChart model

(b) Generated HTML5+JS chart

Fig. 3: Simple OrgChart model and corresponding graphical representation

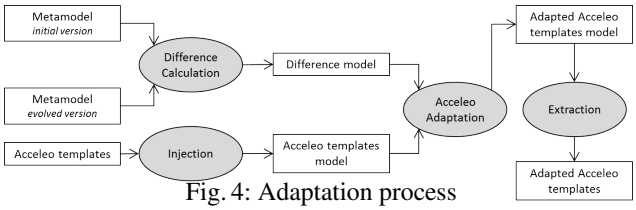


Fig. 4: Adaptation process

3 Adaptation of Acceleo templates

In this section we propose an approach able to adapt Acceleo templates that have to be migrated because of changes operated on the corresponding metamodels. The approach is based on the process shown in Fig 4 and it resembles the techniques we have already applied to adapt ATL transformations [7], TCS specifications [13], and GMF editors [10], that are different kinds of artifacts having the same co-evolution problem. In particular, the adaptation process consists of the following main activities:

▷ *Difference calculation*, given two subsequent versions of the same metamodel, 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 [4];

▷ *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[14];

▷ *Generation of the adapted Acceleo templates*, the differences represented in the difference model are taken as input by the *Acceleo Adapter Transformation* able to adapt the source templates with respect to the operated metamodel modifications.

Concerning the last step of the process it is important to recall that metamodel changes can be classified as follows [6,9]:

▷ *non-breaking changes*: changes that do not break existing Acceleo templates that are still valid with the new version of the metamodel;

▷*breaking and resolvable changes:* changes that affect the validity of existing Acceleo templates but that can be automatically adapted to be applied on models conforming to the new version of the metamodel;

▷ *breaking and unresolvable changes:* changes that affect the validity of existing Acceleo templates and user intervention is required to migrate them.

By considering the previous classification, the adaptation process shown in Fig. 4 is able to migrate Acceleo templates only in case of breaking and resolvable changes. In case of unresolvable changes, comments are added in the generated templates in order to highlight the parts of the templates that have to be fixed by developers. Because of space limitations, in this paper we do not list the catalogue of metamodel changes according the classification above and interested readers can refer to [6] for a detailed discussion. In the remaining of the section we give some details about the representation

[4] EMFCompare: `http://www.eclipse.org/emf/compare/`

VI

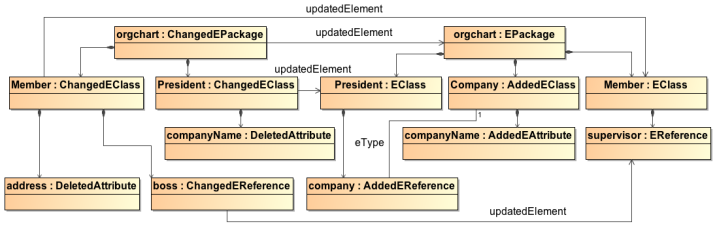


Fig. 5: Delta model representing the differences between the metamodels in Fig. 1

of metamodel differences (Section 3.1) and about the management of some breaking and resolvable changes (Section 3.2).

3.1 Representation of metamodel changes

The differences between different versions of a same metamodel are represented by exploiting the *difference metamodel* concept, presented by some of the authors in [15]. In particular, given two Ecore metamodels, their difference conforms to a difference metamodel *MMD* derived from Ecore by means of a model transformation as follows: 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: *i) additions*, new elements are added in the initial metamodel; *ii) deletions*, some of the existing elements are deleted, and *iii) changes*, some of the existing elements are updated. Fig. 5 shows a fragment of the model representing the differences between the metamodels in Fig. 1. For instance, the renaming of the reference *boss* into *supervisor*, is represented by means of the *ChangedClass* named *Member* that has the *ChangedReference boss* referring by means of the reference to the new *updatedElement* to the reference named *supervisor*.

3.2 Acceleo Adapter Transformation

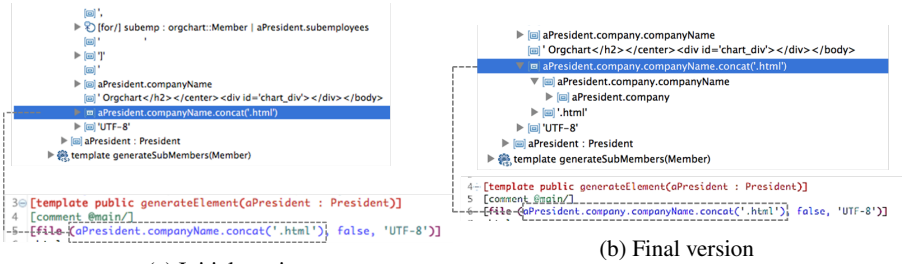
The adaptation of affected Acceleo templates is performed by means of an ATL transformation that takes as input the model of the affected Acceleo templates, the difference model representing the evolution of two subsequent versions of the same metamodel, and generates the adapted Acceleo templates. The transformation consists of a *conservative copy* part, including rules that simply copy the not affected elements in the template, and a *migration* part, consisting of one rule, each devoted to the management of a specific metamodel change. A fragment of the developed transformation is shown in Listing 1.1.Due to space limitations, Listing 1.1 reports only the rules managing the change patterns discussed in Section 2.2. Following we will describe the rules of the *Acceleo Adaptater* showing how the migration has been done on the template of our case study. As can be seen in Fig. 6 the corrupted Acceleo template is injected in a model with extension ".emtl", compliant to the Acceleo metamodel. In Figures 6a, 6b, excerpts of the templates and corresponding models have been reported (top is the model injected from the template source in the bottom).

```
1 module AcceleoAdapter;
2 create OUT:AcceleoMM from IN:AcceleoMM, InitialMM:ECORE, DELTA:DELTAMM, EvolvedMM:
      ECORE;
3 ...
4 rule PropertyCallExpExtractMC {
5  from s : AcceleoMM!PropertyCallExp  (s.existExtractMC())
6  to t : AcceleoMM!"ecore::PropertyCallExp" (
7   ...
8  referredProperty <- s.getReferenceExtractMC(),
9   source <- t1
10 ),
11 t1 : AcceleoMM!PropertyCallExp(
12  source <- thisModule.VariableExpExtractMC(s.source),
13  referredProperty <- s.getReferredPropertyExtractMC()
14 )
15 }
16 lazy rule VariableExpExtractMC
17 {
18  from s : AcceleoMM!VariableExp
19  to t: AcceleoMM!"ecore::VariableExp"(
20      ...
21      referredVariable <- t1
22  ),
23  t1 : AcceleoMM!"ecore::Variable"(..)
24 }
25 rule PropertyCallExp {
26  from s : AcceleoMM!PropertyCallExp
27  (not s.deletedEStructuralFeature() or not s.referredProperty.
       isChangedEStructuralFeature())
28  to t : AcceleoMM!"ecore::PropertyCallExp" (
29  ...
30  referredProperty <- s.getReferredProperty()
31 )
32 }
33 rule PropertyCallExpChanged {
34  from s : AcceleoMM!PropertyCallExp
35  (s.referredProperty.isChangedEStructuralFeature())
36  to t : AcceleoMM!"ecore::PropertyCallExp" (
37  ...
38  referredProperty <- s.referredProperty.getReferredPropertyChanged()
39 )
40 }
41 ...
```

Listing 1.1: Fragment of the Acceleo adapter transformation

PropertyCallExpExtractMC (lines 4-15 in Listing 1.1) is the matched rule responsible for managing attributes involved in extract metaclass changes, like the attribute *companyName* of the metamodel (Fig. 1a), cited in the template highlighted in Fig. 6a line 5. The filter condition in this rule calls the *existExtractMC* helper that checks if he considered attribute is involved in some extract metaclass change pattern. In this specific case, such a change pattern occurs because of the addition of the metaclass *Company*, the deletion of the attribute *companyName*, the addition of the reference *company* and the changed changed metaclass *President*. The execution of such rule on the considered template generates two nested *PropertyCallExp*s (see lines 8-14) for the proper navigation of the extracted attribute, see Fig. 6b, where the expression *"aPresident.company.companyName"* corresponds to a nested element in the related model. The *PropertyCallExpExtractMC* rule calls the lazy rule *VariableExpExtractMC* (see line 12) that creates new *VariableExp* and *Variable* OCL elements. The value of *referredProprerty* (line 13) binding is set by the helper *getReferredPropertyExtractMC* that retrieves the new reference according to the information available in the difference

VIII



(a) Initial version

(b) Final version

Fig. 6: Fragment of the *generateElement* template

model. This rule is able then to migrate the expression *aPresident.companyName* to the new expression *aPresident.company.companyName*.

The matched rule *PropertyCallExp* (line 25-32) is one of the conservative copy rule for the elements in the templates that are not affected by the operated metamodel changes. In fact, the rule condition at line 27 checks if the considered element is not deleted or changed according to the information available in the difference model. At line 30, the value of *referredProperty* is set by *getReferredProperty* that derives the right reference. In our example this rule is responsible for the deletion of the expression *m.address* (line 36 in Fig. 2) related to the deleted attribute *address* in the metamodel.

The matched rule *PropertyCallExpChanged* (line 33-40) manages reference renaming changes as in the case of the reference *boss* renamed as *supervisor* in our metamodels (Figures 1a and 1b). The condition of the rule checks if the renaming is occurring in the difference model, by using the helper *isChangedEStructuralFeature*. In this case the new *PropertyCallExp* is created, setting the *referredProperty* with the new value coming from the renamed element in the difference model, by using the helper *getReferredPropertyChanged*. For instance, the expression "*m.boss*" in Fig. 2 (line 41), referring to the renamed reference, is replaced with the expression "*m.supervisor*".

4 Related Work

Metamodeling ecosystems and coupled evolution have been presented in [12], exploring the problem artefact by artefact and including the relation definition among them. Coupled evolution of models and metamodel has been previously exhaustively treated in [6,5,16]. These works focus on the problem of models migration when metamodel changes. They use a model migration language, or an higher order transformation for migrating models. These approaches use a conservative copy for the non-breaking changes, like in our approach. Obviously the atifact kind is different but the intent is the same. Moreover, transformations and metamodel co-evolution is another interesting topic investigated in [8,17,9]. They propose methods and discussions about the problem that we have changing the metamodel which the transformations refer to. All those works use the similar definitions for the classification of changes. Other kinds of artefacts defined on the top of the metamodel can be concrete syntaxes definitions, like diagrammatic or textual. Also these artefacts have dependencies to the metamodel that need to be restored when the metamodel evolves. In [7] and [10] those problems are respectively treated proposing automation similar to the one described in this paper.

This work is strictly related the OCL definition [18], since the migration part is inherent to the Object Constraint Language used by Acceleo for the model navigation in the templates. The work developed for the migration of acceleo templates can be the inspiration or partially reused in other migration of artifacts using OCL, like OCL Query or OCl expression in ATL and so on. In [19] the authors have dealt with the problem of constraints adaptation in order to be compliant to the evolution of their associated metamodels. Since maintaining OCL constraints can be a tedious task, Kahina Hassam et al. propose to assist the developer to rewrite them after each evolution of the associated metamodels. In [20], the authors presented an architecture to automate coupled evolution on an arbitrary software domain. They compute equivalences and differences between any pair of metamodels (e.g., representing schemas, UML models, ontologies, grammars) to derive adaptation transformations from them, and they apply these adaptations as step wise automatic transactions on the initial metamodel, to obtain the final metamodel. These works are related for the OCL part that is in common with our work. In [21] presents *ChainTracker*, a general conceptual framework, and model-transformation composition analysis tool, that supports developers when maintaining and synchronizing evolving code-generation environments. *ChainTracker*, gathers and visualizes model-to-model, and model-to-text traceability information for ATL and Acceleo model-transformation compositions.

5 Conclusions and future work

The problem of coupled evolution in Model-Driven Engineering represents one of the major impediment for the practice of such software discipline. Several approaches have been already proposed mainly focussing on the adaptation of models, transformations, and - at different extent - editors. This paper extended existing techniques to the adaptation of template-based code generators, because such kind of code generators are widely used and part of routinary practices. In particular, the paper proposed an ATL adaptor to consistently migrate Acceleo templates in accordance to the changes operated on the corresponding metamodel. The main contribution of the paper consists in a) the refactoring coverage which is extensive and considers the major refactorings classified in [11]; and b) the migration of OCL expressions which are used by Acceleo for model traversing. The approach has been implemented and is illustrated throughout the paper by means of a running example. To the best of our knowledge, this is the first attempt in addressing the problem of the coupled evolution of template-based code generators. Future work includes the possibility of covering a part of the breaking and non resolvable cases by introducing models with partiality and uncertainty borrowed from the area of requirement engineering.

References

1. Schmidt, D.C.: Guest NOOPeditor's Introduction: Model-Driven Engineering. Computer **39** (2006) 25–31
2. Di Ruscio, D., Iovino, L., Pierantonio, A.: Coupled Evolution in Model-Driven Engineering. Software, IEEE **29** (2012) 78–84
3. Buckley, J., Mens, T., Zenger, M., Rashid, A., Kniesel, G.: Towards a Taxonomy of Software Change: Research Articles. Volume 17., New York, NY, USA, John Wiley & Sons, Inc. (2005) 309–332

4. Wachsmuth, G.: Metamodel Adaptation and Model Co-adaptation. In Ernst, E., ed.: ECOOP 2007 Object-Oriented Programming. Volume 4609 of LNCS., Springer (2007) 600–624
5. Herrmannsdoerfer, M., Benz, S., Juergens, E.: COPE - Automating Coupled Evolution of Metamodels and Models. In: Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming. Genoa, Berlin, Heidelberg, Springer-Verlag (2009) 52–76
6. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: Automating Co-evolution in Model-Driven Engineering. In: Enterprise Distributed Object Computing Conference, 2008. EDOC '08. 12th International IEEE. (2008) 222–231
7. Di Ruscio, D., Iovino, L., Pierantonio, A.: A Methodological Approach for the Coupled Evolution of Metamodels and ATL Transformations. In Duddy, K., Kappel, G., eds.: Theory and Practice of Model Transformations. Volume 7909 of LNCS., Springer (2013) 60–75
8. Levendovszky, T., Balasubramanian, D., Narayanan, A., Karsai, G.: A Novel Approach to Semi-automated Evolution of DSML Model Transformation. In van den Brand, M., Gaevi, D., Gray, J., eds.: Software Language Engineering. Volume 5969 of LNCS. Springer (2010) 23–41
9. Garca, J., Diaz, O., Azanza, M.: Model Transformation Co-evolution: A Semi-automatic Approach. In Czarnecki, K., Hedin, G., eds.: Software Language Engineering. Volume 7745 of LNCS. Springer (2013) 144–163
10. Di Ruscio, D., Lmmel, R., Pierantonio, A.: Automated Co-evolution of GMF Editor Models. In Malloy, B., Staab, S., van den Brand, M., eds.: Software Language Engineering. Volume 6563 of LNCS. Springer (2011) 143–162
11. The MDE Research Group: The Metamodel Refactorings Catalog. `http://www.metamodelrefactoring.org` (2013) University of L'Aquila.
12. Di Ruscio, D., Iovino, L., Pierantonio, A.: Evolutionary Togetherness: How to Manage Coupled Evolution in Metamodeling Ecosystems. In: Procs.of the 6th Int. Conf. on Graph Transformations. ICGT'12, Berlin, Heidelberg, Springer-Verlag (2012) 20–37
13. Di Ruscio, D., Iovino, L., Pierantonio, A.: Managing the Coupled Evolution of Metamodels and Textual Concrete Syntax Specifications. In: Software Engineering and Advanced Applications (SEAA), 2013 39th EUROMICRO Conf. on. (2013) 114–121
14. Di Rocco, J., Iovino, L., Pierantonio, A.: Bridging State-based Differencing and Co-evolution. In: Procs.of the 6th Int. Workshop on Models and Evolution. ME '12, New York, NY, USA, ACM (2012) 15–20
15. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: A Metamodel Independent Approach to Difference Representation. Journal of Object Technology **6** (2007) 165–185
16. Rose, L., Kolovos, D., Paige, R., Polack, F.: Model Migration with Epsilon Flock. In: Proc. ICMT. Volume 6142 of LNCS., Springer (2010) 184–198
17. Wagelaar, D., Iovino, L., Ruscio, D., Pierantonio, A.: Translational Semantics of a Co-evolution Specific Language with the EMF Transformation Virtual Machine. In: Theory and Practice of Model Transformations. Volume 7307 of LNCS. Springer (2012) 192–207
18. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. 2 edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2003)
19. Hassam, K., Sadou, S., Gloahec, V.L., Fleurquin, R.: Assistance system for OCL constraints adaptation during metamodel evolution. In: Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on, IEEE (2011) 151–160
20. Vermolen, S., Visser, E.: Heterogeneous Coupled Evolution of Software Languages. In Czarnecki, K., Ober, I., Bruel, J.M., Uhl, A., Vlter, M., eds.: Model Driven Engineering Languages and Systems. Volume 5301 of LNCS. Springer (2008) 630–644
21. Guana, V., Stroulia, E.: ChainTracker, a Model-Transformation Trace Analysis Tool for Code-Generation Environments. In Di Ruscio, D., Varr, D., eds.: Theory and Practice of Model Transformations. Volume 8568 of LNCS. Springer International Publishing (2014) 146–153