Specification of Adaptable Model Migrations

Paola Vallejo, Mickaël Kerboeuf and Jean-Philippe Babau
University of Brest/Lab-STICC - MOCS Team, Brest, France

Keywords: Metamodel and Model co-evolution, Migration Specification.

Abstract: This paper puts the focus on adaptable model migrations. A dedicated formalism is introduced to combine automatically-generated migrations with custom-made migrations. To illustrate this issue and the approach we suggest to address it, a prototype engine is presented. Then, the prototype is applied on a case study. The prototype processes the migration specifications that have been automatically generated and then customized. The case study consists of the reuse of a mapping tool, in order to change highlighted places. During the reuse process the migration specification is customized in order to produce different migrated models.

1 INTRODUCTION

Reusing legacy tools allows to reduce the cost of producing the tooling for a Domain Specific Modeling Language. A legacy tool is defined by its specific metamodel (the *tool metamodel*): the tool metamodel contains only the elements needed by the tool to execute its functionalities. A common issue when trying to reuse a legacy tool is that the context in which the tool should be used (the *domain metamodel*) is different to the tool metamodel.

Even if these metamodels are different, we can suppose that they share a common subset of close concepts. Thus, the *tool metamodel* can be considered as *an evolution* of the *domain metamodel*. From this point of view, the reuse of a legacy tool implies to put existing models under the scope of the legacy tool by means of *migration* techniques.

A well known and rather obvious way to achieve this purpose relies on the principle of metamodel and model co-evolution. It basically allows the metamodel transformations to be automatically reflected at the model level. Thus, model-level specifics cannot be automatically taken into account by co-evolution. In order to overcome this lack of model-level customization, we present an approach to combine both automatically-generated model migrations with custom-made model migrations.

This paper is structured as follows. Section 2 introduces a motivating example. Section 3 presents the principles of the *migration specification* which underlies our approach. A case study is presented in section 4 to show the relevance of this approach. Related works are discussed in section 5. The paper is concluded with an outlook on future work in Section 6.

2 MOTIVATION

This section introduces a case used as a running example throughout the rest of the paper. *MapView* is a legacy tool we aim at reusing. It displays a map in which the location of specific buildings is marked. The color and the label of the marker depend on the place's type. The tool has been implemented by using the Google Maps API [1]. An excerpt of the Ecore metamodel of the input data that can be processed by this tool is shown in figure 1. This metamodel introduces the concepts of City, Neighborhood, Place and Address. There are different types of places (e.g. University, Dormitory, HealthCareCenter). Each place is located at a specific Address.

Figure 2 shows a variant of the metamodel on which *MapView* has been designed. This metamodel corresponds to an excerpt of a *City Information System* (CIS) which represents the population of a city. It contains information about the citizens, their job, theirs studies, their accommodation and theirs places in the city. These data are typically expected to be collected during a census.

This metamodel can be seen as a variant of the *MapView* metamodel with additional elements (i.e. Citizen class and all its features, *citizens* reference, *zipCode* attribute and *country* attribute of the City

[1] https://developers.google.com/maps/

Vallejo P., Kerboeuf M. and Babau J..
Specification of Adaptable Model Migrations.
DOI: 10.5220/0005231200320039
In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development* (MODELSWARD-2015), pages 32-39
ISBN: 978-989-758-083-3

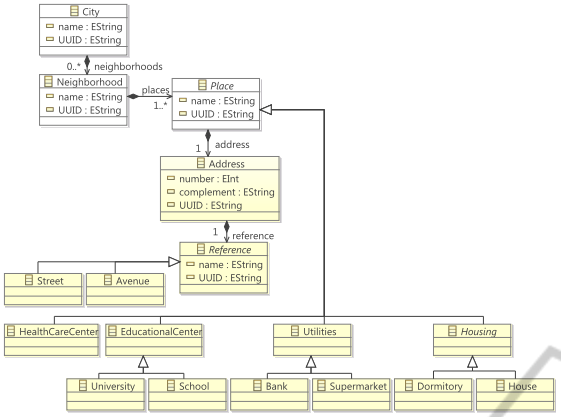

Figure 1: Excerpt of MapView's metamodel.

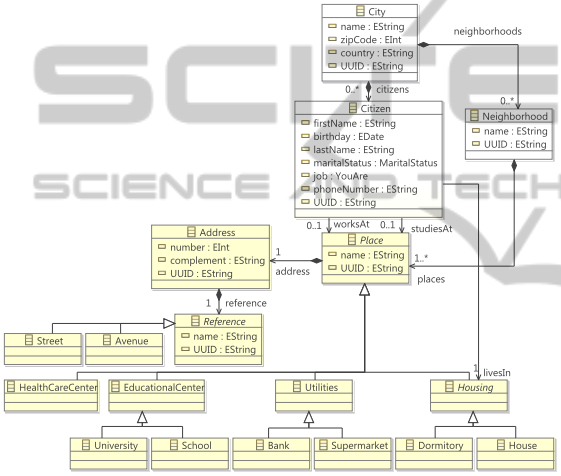

Figure 2: Excerpt of CIS metamodel.

class). The reuse of *MapView* in this context requires the *deletion* of these extra-elements.

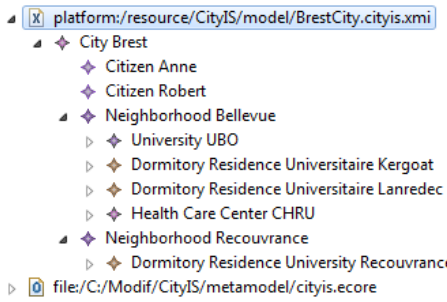

Figure 3: Simple CIS model.

Figure 3 presents a model conforming to the CIS metamodel. To put it under the scope of *MapView*, we apply typical co-evolution operators like *Remove* and *Rename* (Herrmannsdoerfer et al., 2010) .

The *modified metamodel* fully matches with the metamodel of *MapView*. Then, *MapView* can be applied on the migrated model. The migrated model is illustrated by figure 4, citizens Anne and Robert have been removed. The outcome of the tool is depicted by figure 5. The map marks one university (blue marker labeled with U), one health care center (red marker labeled with H) and three dormitories (brown marker labeled with D).

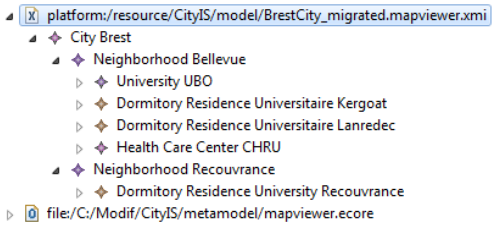

Figure 4: Migrated model.

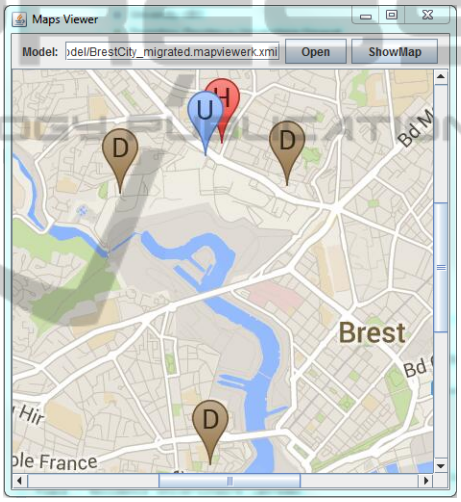

Figure 5: MapView outcome.

In the context of reuse, the same tool can be used according to different and specific needs. For instance, *MapView* will be reused to provide a view of specific places:

1. Only dormitories
2. The university and the dormitories near to it
3. The health care center and the closest dormitory

In case 1, model migration involves deletion of all instances of university and health care center. In cases 2 and 3, some instances of dormitory have to be kept and some others have to be removed. The way to select the dormitories to be kept depends on the notion of nearness.

A way to remove the unnecessary instances is by using co-evolution operators. In co-evolution of metamodel and models, if a class is removed at the metamodel level, all its instances are removed at the model

level. If a class is kept, thus all its instances are kept too.

There are tools as Edapt [2] that allows to manually specifying model migrations when they are so specific to the model context. Edapt generates migration code for instances affected by metamodel transformations. As in cases 1, 2 and 3 there are not metamodel transformations, migration code is not generated, so it is not possible to specify specific model migrations.

In all cases, classes HealthCareCenter, University and Dormitory have to persist at the metamodel level for structural needs (modified metamodel must fully matches with tool metamodel). Co-evolution operators are not useful in these cases because it is not possible to define a model migration for instances whose corresponding class is not transformed.

Another way to handle cases 1, 2 and 3 is by using classical transformation languages, for example ATL [3]. ATL allows to define specific model migrations.

The next code is an excerpt of the ATL code that keep only dormitories.

```
-- @path City=/.../metamodel/cityis.ecore
-- @path Mapview=/.../metamodel/mapviewer.ecore

module city2mapview;
create OUT : Mapview from IN : City;

rule Dormitory2Dormitory {
from
c : City!Dormitory
to
m : Mapview!Dormitory (
name <- m.name
...)
}

rule deleteUniversity {
from
c : City!University
to
drop
}
```

The next code is an excerpt of the ATL code that keep only dormitories located near to the university.

```
-- @path City=/.../metamodel/cityis.ecore
-- @path Mapview=/.../metamodel/mapviewer.ecore

module city2mapview;
create OUT : Mapview from IN : City;

rule Dormitory2Dormitory {
from
c : City!Dormitory(
      c.address.reference.name  = 'Lanredec'
      || c.address.refrence.name = 'Archives')
to
m : Mapview!Dormitory (
name <- m.name
...)
}
```

The next code is an excerpt of the ATL code that keep the dormitory close to the health care center.

```
-- @path City=/.../metamodel/cityis.ecore
-- @path Mapview=/.../metamodel/mapviewer.ecore

module city2mapview;
create OUT : Mapview from IN : City;

rule Dormitory2Dormitory {
from
c : City!Dormitory(
     c.address.reference.name  = 'Lanredec' )
to
m : Mapview!Dormitory (
name <- m.name
... )
}
```

The difference between the three pieces of code is the constraint *c.address.reference.name* = . In the first case, there is not constraint because all instances of dormitory are kept. It is mandatory to specify the migration code for each case, which makes this solution model dependent, programming language oriented and not generic.

More generally, three cases can be encountered: 1) A class and all its instances have to be removed. For example, Citizen class. 2) A class is kept but *some* of its instances have to be removed. For example, Dormitory in cases 2 and 3. 3) A class is kept but all its instances have to be removed. For example, University in case 1.

In those cases, there are two concerns, the first one, related to the classes; the second one, related to model instances and specific contexts. There is a lack of a mechanism to ensure the separation those of concerns.

Then, we propose to address the proposed cases by an approach able to generate model migrations and allowing customization of migrations without making modifications directly in the migration's code.

It consists in promoting the *adaptation* of automatically generated model migration thanks to a *formal* and *generic migration specification*. For a given metamodel MM transformed into a new metamodel MM', and for a given model $\mathfrak{m}$ conforming to MM: instead of *directly reflecting* the metamodel evolution at the model level by means of a generated migration mig such that mig($\mathfrak{m}$) provides a migrated model conforming to MM', we suggest to generate a *migration specification* $\overrightarrow{\mathfrak{m}}$ such that provided to a dedicated and

[2] http://www.eclipse.org/edapt/

[3] http://www.eclipse.org/atl/

generic engine M, it produces the expected migration mig (*i.e.* $\mathsf{M}(\overrightarrow{\mathfrak{m}}) = \mathsf{mig}$). As a consequence, the migration specification $\overrightarrow{\mathfrak{m}}$ is *editable* and *processable*. Even if it is automatically generated, it can be *modified* before being processed by the generic engine M. Then we can obtain an *adapted migration* mig.

The next section states the formal basis of this approach. The following section illustrates this approach with the reuse of *MapView*.

3 FORMAL FRAMEWORK FOR ADAPTABLE MODEL MIGRATIONS

A *migration specification* relies on *formal model denotations*, which are *oriented and labeled graphs*. These notions are detailed in the first part of this section. They underlie the prototype presented in the second part.

3.1 Migration Specification Foundations

Model Graph. A model graph is a *graph-based denotation* of a model, *i.e.* a labeled graph composed of *vertices* denoting *instances* and *scalar values*, and *edges* denoting *references* and *attributes*. The *namespaces* corresponding to instances, scalar values, attributes and references are defined by the following *alphabets*, (*i.e.* non-empty finite set of symbols):

I : instances, S : scalar values, A : attributes, R : references.

We call *model* and note $\mathfrak{m}$ a triplet composed of a set of instances corresponding to vertices noted V, and two sets of edges noted E_a and E_r. The first set of edges denotes attributes. It relates instances to scalar values through attribute names. The second one denotes references. It relates instances to each other through reference names:

$$\mathfrak{m} \triangleq \left(V \, , \, E_a \, , \, E_r \right) \quad \text{with} \begin{cases} V & \subseteq & I \\ E_a & \subseteq & V \times A \times S \\ E_r & \subseteq & V \times R \times V \end{cases}$$

We note $\mathfrak{m}.V$, $\mathfrak{m}.E_a$ and $\mathfrak{m}.E_r$ the V, E_a and E_r components of a given model $\mathfrak{m}$.

As an illustration, figure 6 depicts an excerpt of the CIS model with this graph representation.

Migration Specification. For a given model $\mathfrak{m}$, we call *migration specification* and note $\overrightarrow{\mathfrak{m}}$ a quadruplet composed of $\mathfrak{m}$ and of three sets noted D_i, D_a and D_r. These sets specify the instances, attributes and references of $\mathfrak{m}$ that are intended to be *removed*:

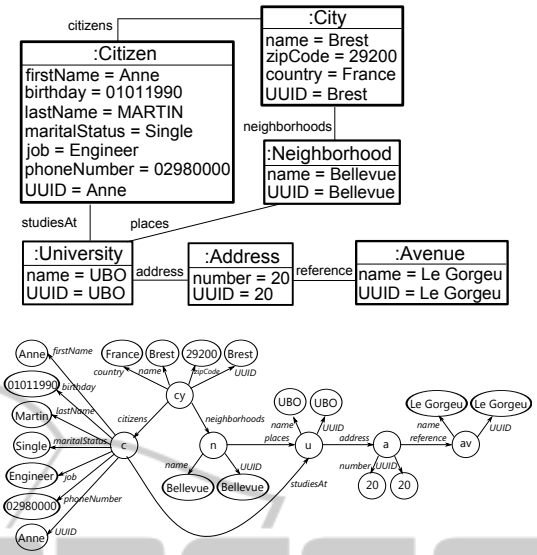

Figure 6: Abstract syntax and semantics.

$$\overrightarrow{\mathfrak{m}} \triangleq \left(\mathfrak{m}, D_i, D_a, D_r \right) \text{ where } \begin{cases} D_i & \subseteq & \mathfrak{m}.V \\ D_a & \subseteq & \mathfrak{m}.V \times A \\ D_r & \subseteq & \mathfrak{m}.V \times R \end{cases}$$

We note $\overrightarrow{\mathfrak{m}}.D_i$, $\overrightarrow{\mathfrak{m}}.D_a$ and $\overrightarrow{\mathfrak{m}}.D_r$ the D_i, D_a and D_r components of a given migration specification $\overrightarrow{\mathfrak{m}}$. The migration specification is intended to be used together with the model to produce a *migrated model*. This migration specification is *computed* from coevolution (i.e. *Modif specification* (Babau and Kerboeuf, 2011)). A specification makes it possible to state the deletion of a class, an attribute or a reference (corresponding to D_i, D_a and D_r at the model level).

Migration Engine. We call *migrator* and note M the tool aiming at producing a *migrated model* from a *migration specification*. According to the following algorithm, it commits the *deletion* of instances and of features (attributes and references) on the source model to produce a target model:

$$\begin{array}{l} \mathfrak{m}' = \mathsf{M}(\overrightarrow{\mathfrak{m}}) \\ \mathfrak{m}'.V = \mathfrak{m}.V \setminus \overrightarrow{\mathfrak{m}}.D_i \\ \mathfrak{m}'.E_a = \mathfrak{m}.E_a \setminus \{(i,a,s) \in I \times A \times S \mid i \in \overrightarrow{\mathfrak{m}}.D_i \vee (i,a) \in \overrightarrow{\mathfrak{m}}.D_a\} \\ \mathfrak{m}'.E_r = \mathfrak{m}.E_r \setminus \{(i,r,i') \in I \times R \times I \mid i \in \overrightarrow{\mathfrak{m}}.D_i \vee i' \in \overrightarrow{\mathfrak{m}}.D_i \vee (i,r) \in \overrightarrow{\mathfrak{m}}.D_r\} \end{array}$$

As an illustration, we note $\mathfrak{m}$ the model of figure 6 and $\overrightarrow{\mathfrak{m}}$ its migration specification aiming at producing a model conforming to the a metamodel of figure 1, *i.e.* a model in which all instances of Citizen have

been removed. This specification is formally and explicitly defined by its components:

$$\overrightarrow{\mathfrak{m}}.D_i = \{c\} \qquad \overrightarrow{\mathfrak{m}}.D_a = \{zipCode, country\}$$
$$\overrightarrow{\mathfrak{m}}.D_r = \{studiesAt, citizens\}$$

Once the migrator has been applied to $\overrightarrow{\mathfrak{m}}$, we obtain the migrated model $\mathfrak{m}_\mathsf{m}$ illustrated by figure 7.

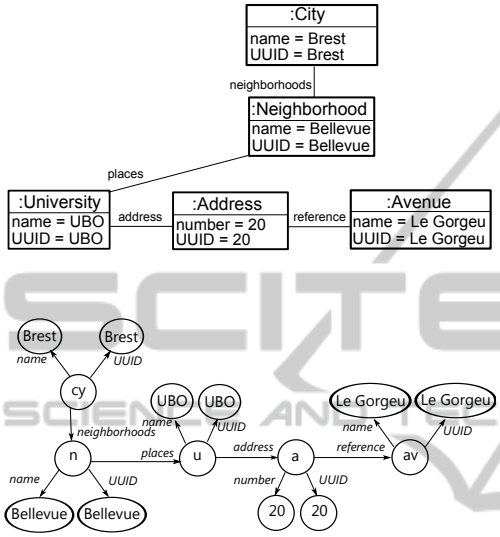

Figure 7: Abstract syntax and semantics of a migrated model without Citizen.

The migration is performed at the model level. It does not require any *metadata*.

3.2 Migration Specification Implementation

A *migrator* prototype based on Ecore and Modif (Babau and Kerboeuf, 2011) has been developed according to the formal principles of *migration specification*. The refactoring, the migrator and the tool to be reused form a toolchain depicted by figure 8 and detailed as follows.

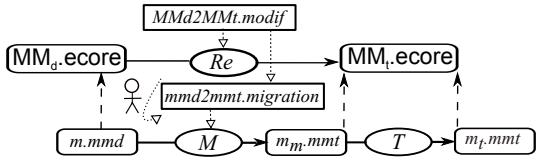

Figure 8: Typical toolchain including the *migrator*.

Refactoring (Re). The first step is the application of co-evolution operators at the metamodel level. Consider a domain metamodel MMd.ecore in figure 8. On it, co-evolution operators are applied to obtain an evolved metamodel that matches with the tool metamodel (MMt.ecore). The prototype uses a *Modif Specification* to indicate the operators to be applied, but some other metamodel transformation tool or language can be used as well (e.g. COPE, ATL, QVT).

Migration Specification Generation. The second step is the automatic generation of a *by default* migration specification conforming to the metamodel of figure 9. The *by default* migration specification is derived from the co-evolution operators applied at the Refactoring step. It contains URIs of source and target models and metamodels. The instances that must be updated by the migration are identified by a unique identifier (UUID). When a class is removed at the metamodel level, deleteInstance is set to *true* for each Instance of the removed class. Otherwise, it is set to *false* and then, the modification is related to *features*.

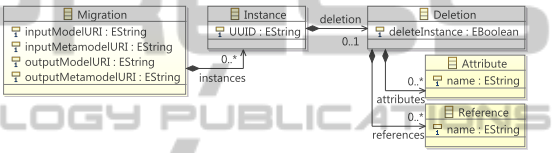

Figure 9: Graphical view of Migration metamodel.

The migration specification is typically generated from Modif, but it can be defined from other sources, including a model editor generated from the metamodel of figure 9.

Specification Edition. The third step is the edition of the migration specification. In order to take into account some model specifics in the generated migration specification. It is therefore possible to produce migrated models according to specific needs without having to generalize it at the metamodel level. And without modifying any source code.

Migration Engine (M). In the fourth step, the Migration Engine is executed to obtain a migrated model according to the modifications indicated in the migration specification. It takes an MMd input model (m.mmd) and produces a migrated MMt output model (mm.mmt). The Migration Engine is independent of the input model. It means it can be reused for different input models. And it is not overloaded with unnecessary metadata.

Use of a Legacy Tool (T). Before reusing the legacy tool, the migrated model coming from the migrator is validated. Finally, the legacy tool is applied on the migrated validated model.

4 EXPERIMENT

We present in this section the results of the application of the prototype to the *MapView* case study introduced in section 2.

An excerpt of the Modif Specification is illustrated by the next code:

```
root cityis to mapview
Prefix cityis to mapview
URI "file:/C:/Modif/cityisK.ecore"
 to "file:/C:/Modif/mapview.ecore"

class {
     City {
        remove ref citizens
        remove att zipCode
        remove att country
     } ;
     remove Citizen {
        remove att firstName
        remove att birthday
        remove att lastName
        remove ref livesIn
        remove att maritalStatus
        remove att job
        remove ref worksAt
        remove ref studiesAt
        remove att phoneNumber
        remove att UUID
     } ;
     ...
}
```

The features *citizens*, *zipCode* and *country* of the class City are removed. The class Citizen and all its features are removed too.

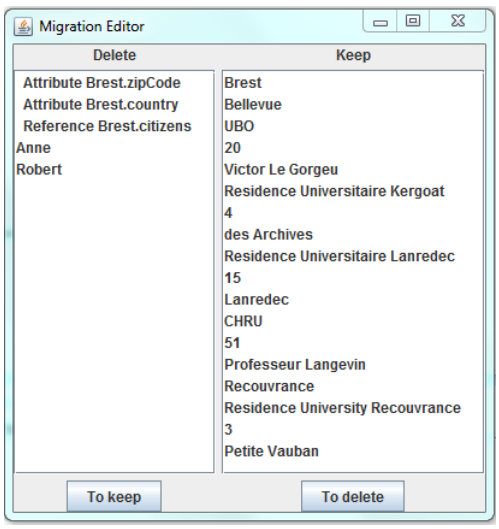

Figure 10: By default migration specification.

Figure 10 depicts the *by default* migration specification generated from the Modif Specification and the model of figure 3. The left column contains the instances and the features that have to be removed. The right column contains the instances that have to be kept.

Once the *by default* migration is generated, it can be edited allowing the specification of customized model migrations. Just by selecting one instance of the *Keep* column and clicking on the *To delete* button, it will be placed on the *Delete* column.

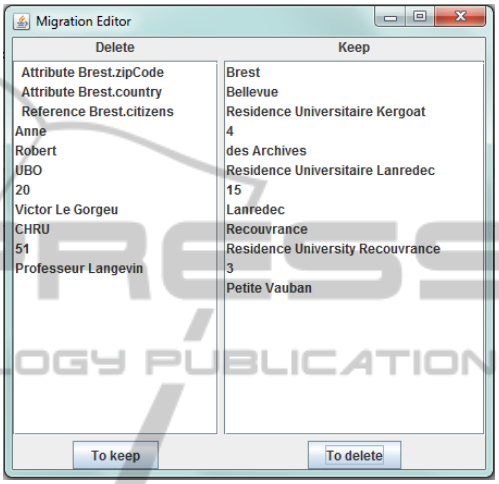

Figure 11: Customized migration specification 1.

Figure 11 presents the migration specification allowing to keep only dormitories (case 1 of section 2).

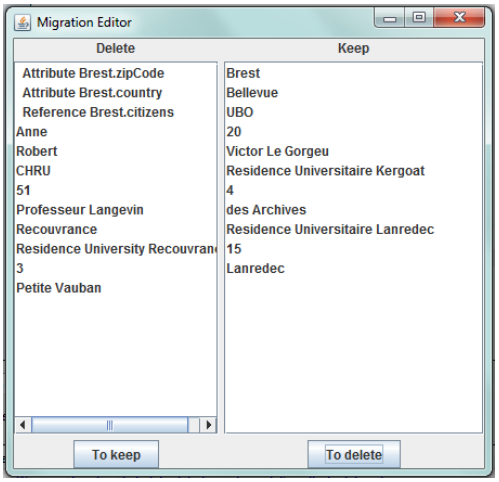

Figure 12: Customized migration specification 2.

Figure 12 illustrates the migration specification that allows to to keep only the university and the dormitories near to it (case 2 of section 2).

Figure 13 illustrates the migration specification allowing to to keep the health care center and the closest dormitory (case 3 of section 2).

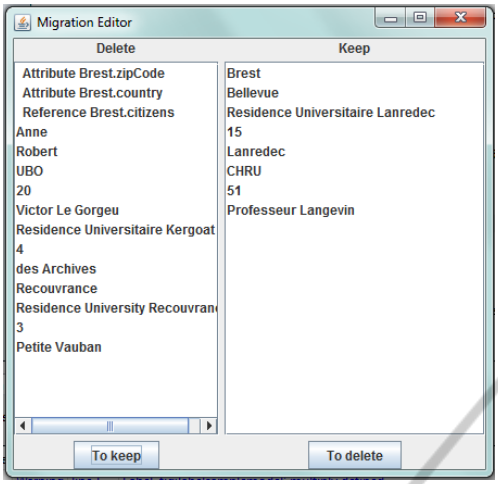

Figure 13: Customized migration specification 3.

From a model and the *by default* migration specification, some *customized* migration specifications can be produced. After customizing the migration specification, the Migration Engine is executed to produce a model conforms to the legacy tool metamodel. Finally, *MapView* is applied. Figure 14 illustrates the specific outcome when the migration specification is that of figure 12.

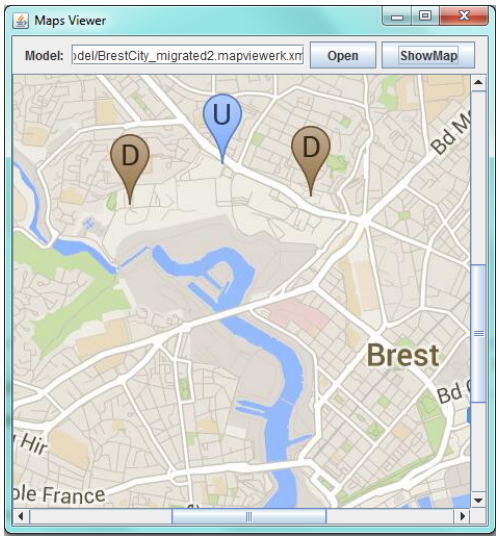

Figure 14: D near to the U.

Our approach is able to generate model migrations and allows customization of migrations. Thanks to the migration editor, a same migration specification can be used as a common basis to generate quickly and efficiently various migrated models. Any knowledge about transformation or programming languages is required to customize model migrations.

5 CURRENT AND RELATED WORKS

The necessity of providing automatic co-evolution of metamodels and models was identified as an important challenge in software evolution by Mens and al. (Mens et al., 2005). With the growing importance of model transformations, much work is intended to improve the definition and the verification of such transformations by providing adapting formalism and tooling.

Wachsmuth (Wachsmuth, 2007) proposes a transformational approach to assist metamodel evolution by stepwise adaptation. Ciccheti et al. (Cicchetti et al., 2009) present a classification of metamodel and model changes. They propose an approach for performing co-evolution automatically, based on dependencies between different kinds of modifications. In those works the migration of models is based on reusable operators, automated and non adaptable. It allows to reduce the effort associated with building a model migration. As a complement, we allow to define customize model migrations by means of a *Migration Specification*. It is possible to generate it from the history of co-evolution operators applied to transform a metamodel.

The authors of (Regg et al.,) and (Agrawal et al., 2003) evoke the need of methods and tools to automate and speed up the process of migrating models. However, some metamodel changes require information during migration which is not available in the model, these migration inherently cannot be fully automated. They enable user interaction during migration only when missing information has to be provided. Epsilon Flock language (Rose et al., 2010) uses a migration strategy that can be extended by adding constraints. COPE (Herrmannsdoerfer and Ratiu, 2009) provides a library of reusable co-evolution operators which produces model migrations automatically. Only in case no reusable operators are available, the user defines custom operators by manually encoding a model migration. When using those approaches, additional effort is necessary to learn a new language for model migrations. In general, the definition of custom migrations is tedious and error-prone. Unlike those approaches, our approach allows user interaction even if migration is fully automated. Definition of new operators is not needed and knowledge about a particular language is not required.

MOLA (Kalnins et al., 2004) is a graphical model transformation language whose main goal is to describe model transformations in a natural and easy readable way. It focuses on easy readability and customization of transformations. This approach is not

related to co-evolution, it means that they not include the notion of co-evolution operators and it works only at the model level.

6 CONCLUSION AND FUTURE WORKS

The paper outlines an approach for supporting the co-evolution of metamodels and models. The proposition combines the automatic generation nature of co-evolution approaches with user interactions. In this way, part of the definition of model migrations is achieved automatically, but it still possible update them manually. We experimented the approach by performing adaptable model migrations combining automatically-generated migration with custom-made migrations. Those model migrations allows to reuse the *MapView* tool according to specific user needs.

Our perspective is to extend the approach to perform adaptable model migrations with others co-evolution approaches. We are working on the definition of other co-evolution operators at model level (change value) and on to define valid model migrations.

REFERENCES

Agrawal, A., Karsai, G., and Shi, F. (2003). Graph transformations on domain-specific models.

Babau, J.-P. and Kerboeuf, M. (2011). Domain Specific Language Modeling Facilities. In *5th MoDELS workshop on Models and Evolution*, Wellington, New Zealand.

Cicchetti, A., Ruscio, D. D., and Pierantonio, A. (2009). Managing dependent changes in coupled evolution. In Paige, R. F., editor, *ICMT*, volume 5563 of *Lecture Notes in Computer Science*. Springer.

Herrmannsdoerfer, M. and Ratiu, D. (2009). Limitations of automating model migration in response to metamodel adaptation. In *MoDELS Workshops*.

Herrmannsdoerfer, M., Vermolen, S., and Wachsmuth, G. (2010). An extensive catalog of operators for the coupled evolution of metamodels and models. In *SLE*.

Kalnins, A., Barzdins, J., and Celms, E. (2004). Model transformation language mola. In *Proceedings of MDAFA (Model-Driven Architecture: Foundations and Applications*.

Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., and Jazayeri, M. (2005). Challenges in software evolution. In *Proc. 8th IWPSE*. IEEE.

Regg, U., Motika, C., and von Hanxleden, R. Interactive transformations for visual models.

Rose, L. M., Kolovos, D. S., Paige, R. F., and Polack, F. A. C. (2010). Model migration with epsilon flock. In Tratt, L. and Gogolla, M., editors, *ICMT*, volume 6142 of *Lecture Notes in Computer Science*. Springer.

Wachsmuth, G. (2007). Metamodel adaptation and model co-adaptation. In *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP'07)*, Lecture Notes in Computer Science. Springer-Verlag.