A DSML for reversible transformations

Mickaël Kerboeuf
LISyC, UBO, UEB
Brest, France
kerboeuf@univ-brest.fr

Jean-Philippe Babau
LISyC, UBO, UEB
Brest, France
babau@univ-brest.fr

ABSTRACT

In this paper, we investigate a way to promote the reuse of legacy tools (or transformations) in specific contexts (defined by specific metamodels). More precisely we suggest a model transformation approach to achieve this purpose.

We first introduce a language based on a metamodel called Modif in order to specify the differences between two semantically close metamodels. We can generate automatically data migration components from a Modif specification. They enable to put data complying with the specific context under the scope of the legacy tool. But more importantly in the case of a rewriting tool, they enable to put the tool's outcome back into the original specific context.

Then we propose a process and a set of helpers based on Modif to automate the reuse of legacy tools for domain-specific contexts. To illustrate this approach, we apply it to the case of simple finite state machines.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Programmer workbench*; D.2.13 [**Software Engineering**]: Reusable Software—*Reuse models, Domain engineering*

General Terms

Design, Languages

Keywords

DSML, tool support, model transformation, code generation, reusable tools

1. INTRODUCTION

Making available tool support for Domain Specific Modeling Languages (DSML) is a fundamental issue for DSML designers. The reuse of a legacy component is a basic way to significantly lower the cost of obtaining the complete tool support for a DSML [4]. The use of a metamodeling environment is highly likely to lower the cost of obtaining these facilities thanks to its generative abilities. It can produce domain-specific editors or browers for metamodel instances. In some cases, it can also offer an environment for domain-specific rules of code generations [16].

But in spite of these available facilities, a DSML designer still needs to add domain-specific components either by specifying them completely, or by coding them directly. In many cases, these specific components are already available but they are designed for *variants* of the metamodel on which the designer is working. For instance, let us consider, on one hand, statecharts complying with a specific metamodel dedicated to formal analysis, and on the other hand, a legacy tool designed for UML statecharts edition. We can properly suppose that they share semantically close constructs. Thus, it is possible to set links between UML concepts and specific formal concepts. It is therefore possible to translate directly UML instances complying with what is expected by the editor into specific statecharts for formal analysis.

Such kind of transformations can be seen as model migration tools [17]. We introduced a language called Modif [1] in order to target easily this kind of transformations. It is a language dedicated to the specification of differences between two variants of semantically close metamodels. The available tool support for this language enables to generate data migration components.

Thus, Modif can be used to put data under the scope of the legacy tool a designer aims at reusing. But in the case of a legacy *rewriting* tool (*i.e.* an *endogenous* transformation), the designer faces a more challenging problem: how the data that have been translated and then modified by the rewriting tool can be translated back into their original context? In this paper, we present a relational mechanism based upon Modif to address this problem and thus to promote the reuse of rewriting tools on different specific domains.

The paper is organized as follows. In the next section, we introduce in details the background of this work and our motivations. Then we present and illustrate the proposition we make to put back the tool's outcome into its specific application domain. The related works are discussed in the following section. To conclude, we present the current and future developments of this work.

2. BACKGROUND AND MOTIVATION

We introduce here the problem of reuse we address. As the solutions we investigate are based upon Modif, a large part of this section is devoted to this language (see [1] for more details). The last point of this section introduces the specific problem of rewriting on which we focus in this paper.

2.1 Reuse of existing tools

As depicted in figure 1, a well-attested and rather obvious way to reuse existing model transformations (*legacy tool*) consists in creating a transformation whose target matches with the metamodel of the input data that are expected by the tool. The source of the transformation is the specific context (a metamodel) where the need for the legacy tool has been identified. The outcome of the tool is supposed to be what is expected in the specific context. Thus, there is no need to rewrite the tool for our specific context. It is obtained by transitivity. We call *injection* the transformation whose purpose is to put data under the scope of a tool.

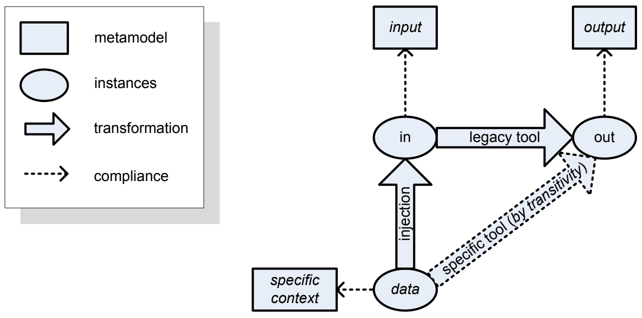

Figure 1: reuse of a tool by model transformation

2.2 A DSL for injections

From a model driven point of view, injection is a model transformation. A suitable way to specify such kind of transformation is to use a specific language like ATL [9], Kermeta [13] or QVT [15].

The specific context and the input metamodel shown on figure 1 are supposed to share close concepts. In light of this finding, we introduced a specific language based on a metamodel called Modif [1] to quickly generate injections. Modif enables to focus efficiently on elementary transformations between two variants of a same metamodel. It is inspired by some functions of persistent storage (CRUD) from a source metamodel to produce a target metamodel. Namely, these functions are *update* (*i.e.* *rename* and *change*) and *deletion*. Modif enables to state simple refactoring operations applied to an input metamodel. To handle Modif, a translator has been developed in Kermeta. It takes a metamodel and produces a new metamodel in accordance with a Modif specification. The instances complying with the first metamodel can be translated into instances complying with the new one. To achieve this translation, a specific tool can be *generated* from the Modif specification.

Figure 2 illustrates the use of Modif to produce injections. Modif is intended to help a designer to map efficiently his own metamodel to an existing metamodel for which an interesting tool has been designed. If this designer aims at reusing this tool, we can properly suppose that at least a subset of his own metamodel has a semantics domain which is equivalent to the targeted metamodel, and that he does not have to create new specific data to match the requirements of the targeted tool. For instance, if the targeted tool is a statechart analyzer built upon a metamodel for finite state machines (FSM), it is rather natural that someone who needs this tool has his own specific structure for FSM (*e.g.* sates as nodes and transitions as edges). It is also

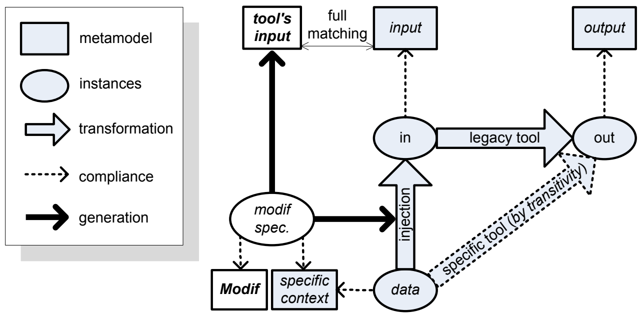

Figure 2: Generation of transformations with Modif

natural that this specific structure for FSM will contain at least all that is needed by the tool the designer expects to reuse. This last point explains why Modif does not involve *creation* operations.

2.3 Process and tool support for Modif

Modif enables to create a target metamodel. This can seem unsuitable for our need to reuse tools built upon legacy metamodels. Actually, in a typical process involving Modif, the generated metamodel is used to check the full matching with a legacy metamodel.

We illustrate this point with the example of a tool built upon a metamodel of finite state machines (FSM). We want to apply it on UML statecharts [14].
The first step of the process is the *generation* of a Modif specification *with default values* for the UML metamodel. Many generators of Modif specifications with various default values are available [1]. In our example, we need to translate UML specifications into FSM specifications. The FSM metamodel is much less expressive than the full UML metamodel. So we need to delete a significant part of UML to match FSM. Then we choose to generate a Modif specification that will delete everything *by default*.
The second step of the process is the *update* of this *by-default* Modif specification so that we keep only what is relevant for the FSM metamodel. Then we produce the target metamodel and we check it fully matches the existing FSM metamodel.
The last step of the process is the *generation* of the translator from UML statecharts into FSM complying *by construction* with the right metamodel.

Modif enables to avoid the tedious tasks of deleting what is not needed, and copying what is needed with elementary refactoring operations like renaming. Its most original function is the ability to *hide* inheritance. References to hidden super classes are spread over the sub-classes. The graph-based denotational semantics of Modif on which we are working formally states its meaning. But all the technical details of Modif and its semantics issues are out of the scope of this work and will be discussed in another paper.

2.4 Reusability of tools' outcome

So far, besides Modif, this way of reusing legacy tools by model transformation is not very original. But we introduce now a more challenging issue. In some cases, the tool's outcome has to be modified to match with the initial specific context. This is typically the case when the tool to be reused is a *rewriting tool.* Input and output data of such tools both comply with a same metamodel. For instance, an optimizer performing constant folding on abstract control flow graphs could be reused on a given specific procedural language. But the optimization performed at an abstract level has to be replaced in the context of the procedural language.

We call *contextualization* the transformation whose purpose is to put back outcome data of a legacy tool into the specific context (metamodel) where the tool is reused. This transformation is a kind of reverse injection. It raises the problem of *reversibility* of transformations.

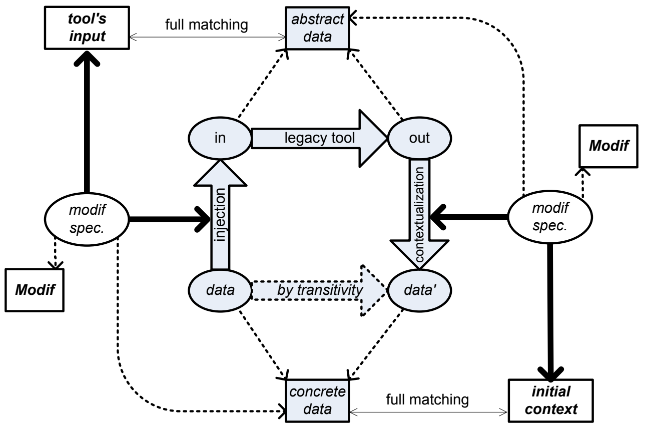

Figure 3: Contextualization of a tool's outcome

Figure 3 depicts an outline of the models and transformations that are involved in this approach. A rewriting tool defined on an abstract level can be reused at a concrete level if we can:
1) put concrete data under the scope of the tool (injection),
2) and translate the tool's outcome into data complying with the initial metamodel (contextualization).

As we mentioned before, the subset of CRUD operations Modif offers does not involve creation for now. Thus, as far as deletion is not needed, a Modif specification can be easily *inverted* without loss. Indeed, in this case, the source and target metamodels specify the same concepts but they are *named* and *structured* differently.

A tool performing Modif inversions has been developed. Two transformations can be generated from a Modif specification and its inverted version. We prove the composition of those two transformations does produce *identity.*

Unfortunately, deletion is typically useful in the context of tools' reuse. Indeed, the target metamodel is restricted to a specific use. It is a *tool*'s definition domain. Thus it embeds less dedicated concepts than the source metamodel. Thus we need an appropriate solution to maintain as far as possible the initial context of data that have been injected within the scope of a rewriting tool. This is the main problem on which we focus in the rest of this paper.

3. KEYS OF DATA RECOVERY

We present now a solution to take into account deletion. We aim at recovering as far as possible deleted data after the application of a legacy tool. In this section, we present and illustrate our proposition.

3.1 Storage and recovery of deleted data

The basic idea of our proposition is quite obvious: if we want to recover data that have been deleted, we have to back them up. Hence the first step: injection has to produce two sets of instances. The first one comes from the execution of basic refactoring operations stated in the Modif specification (including deletion). The second one corresponds to the initial instances *annotated* with labels. A label plays the role of a *key* of relational databases.

These annotated instances comply with a variant of their initial metamodel. In this variant, an attribute called *keyForModif* has been added to each class (which is itself modeled as an *EClass* from the Ecore metamodel [3] we use). This metamodel can be easily generated as also the translator that adds unique keys on instances.

At this stage, the rewriting tool to reuse is applied on the first set of instances. Then this set is updated. We find two kinds of instances in it: instances that already existed before the tool's application and that may have been updated (*i.e.* new reference or attribute values), and new instances created by the tool.

Here comes the second step of our proposition. To achieve the contextualization, we take into account the tool's outcome together with the initial instances annotated with keys. We perform a relational natural join between them. Thus, instances that already existed are reconnected to instances that had been deleted. Of course, new instances are not reconnected to any deleted instances.

This last point is important: it is not possible to define *automatically* a *by default* valid context for instances that have been *created* by the legacy tool. Injection and contextualization are not intended to produce automatically a *complete* specialized prototype from an existing tool. If the rewriting consists in recreating all instances, then these instances cannot be automatically connected to any original context. In this case, either instances that have been backed up are lost, or specific user code has to be added to the generated contextualization in order to make valid initializations. In all cases, our proposition improves the efficiency of the development. Either all the transformations are automatically generated, or small adaptions are necessary on contextualization. As for injection, it is fully generated without modification and the legacy tool it targets can be actually reused.

Figure 4 illustrates the use of Modif with keys. The generators have been updated to take into account keys, especially during contextualization.We currently formalize the joint algorithm to prove that the composition of injection and contextualization produces *identity* even with deletion. For now, we just give here the main stages of this algorithm. Like in figure 4, we call out the set of instances that have been processed by the tool. We call data+key the original instances annotated with keys. The algorithm has 4 stages:

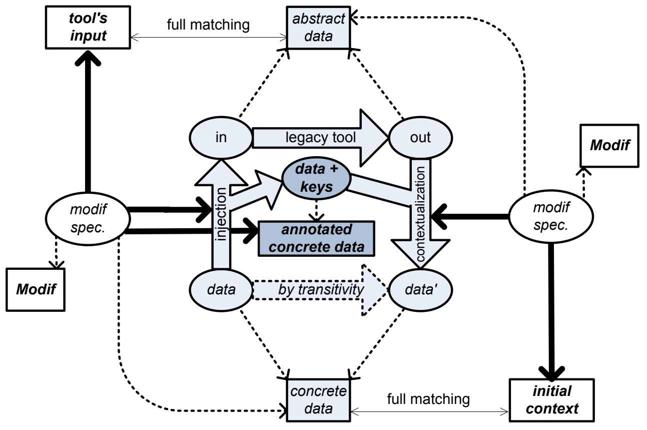

Figure 4: Contextualization with *keys*

1. For each instance of out fitted with a key (*i.e.* that appears in the original context), we look for the unique instance of data+key fitted with the same key. Attributes that had been marked *deleted* in the Modif specification are added in the instances of out.
2. Each instance of data+key whose class had been marked *deleted* in the Modif specification is added to out.
3. For each *reference* of each instance of out we obtain at this stage, if itself or its source or its target had been marked *deleted* in the Modif specification, then it is added in the instances of out.
4. Keys are removed; the inverted Modif is applied. We get instances complying with the original context.

3.2 Application on finite state machines

We present here a case study to underline the benefits that can be reaped of our tools.

Legacy tool on FSM. We defined a *flattener* tool on hierarchical finite state machines (FSM). The Ecore metamodel of input data expected by this tool is shown in figure 5.

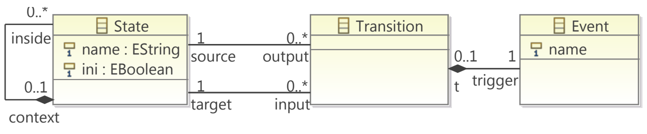

Figure 5: Metamodel of hierarchical FSM

This small metamodel introduces the concepts of *state*, *transition* and *event*. Hierarchy comes from the ability of a state to contain other states. This metamodel does not take concurrency into account. This metamodel is *tool oriented*. Only the data that are strictly required for the flattening are represented.

The flattener we defined on this metamodel produces instances where each state is a *simple* atomic state (*i.e.* it does not contain any other state). The outcome is semantically equivalent to the input FSM. For each super-sate, all incoming transitions are forwarded to the initial substate, and all outgoing transitions are duplicated on each substate.

Another FSM metamodel. The flattener plays the role of a legacy tool we want to reuse on another FSM metamodel. Figure 6 is a variant of the FSM metamodel on which the flattener has been designed.

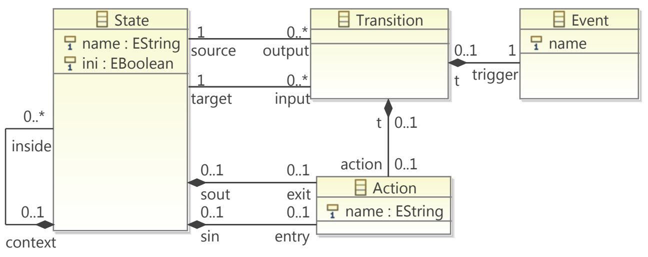

Figure 6: Metamodel of FSM with actions

For the purpose of our example, we willfully introduced a very simple variant of the FSM metamodel. It is a simplified version of UML statecharts, and it has actually been derived from UML using Modif. It has the advantage of highlighting the problem we address. Indeed, in this variant, the only notion of *action* has been added, and it has to be *deleted* if we want to reuse the flattener. But once flattened by a tool that does not take actions into account, the FSM has to be rebuilt with actions.

We defined a FSM model complying with the metamodel of figure 6. It is depicted by figure 7. The state called running contains substates and then it has to be flattened. Actions appear associated to transitions and to states. They won't be taken into account by the flattener.

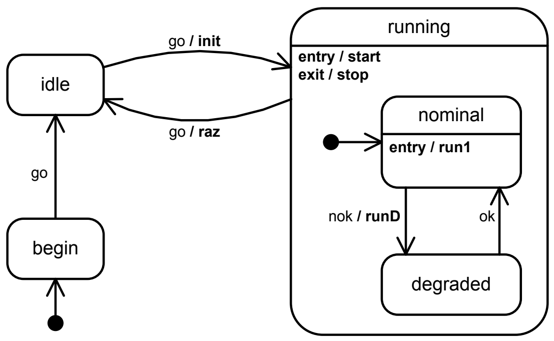

Figure 7: A simple FSM with actions (in bold)

Injection. To begin with, a Modif specification is *generated* for the FSM metamodel involving actions (see figure 6). This Modif specification contains default values. They state everything in the metamodel remains unchanged. Then this Modif specification is updated so that the EClass named Action has to be deleted.

From this Modif specification, we generate the target metamodel (to make sure it fully matches the FSM metamodel handled by the flattener), and the program corresponding to injection. Then we apply the generated injection to the set of instances depicted by figure 7. We obtain 2 FSM.

The first one complies with the FSM metamodel handled by the flattener: it corresponds to the original FSM except that all actions have been removed; the outcome model is the same as figure 7 except all actions including entry and exit are removed.

The second one complies with the original metamodel annotated with keys: each state, transition, event and action has a new and unique attribute called *keyForModif*; the outcome model is the same as figure 7 except that it is enriched with 21 keys (one for each concept: 5 states, 5 transitions, 5 events, 6 actions).

Flattener. At this stage, the flattener can be applied on the first generated FSM. We obtain the flat FSM depicted by figure 8.

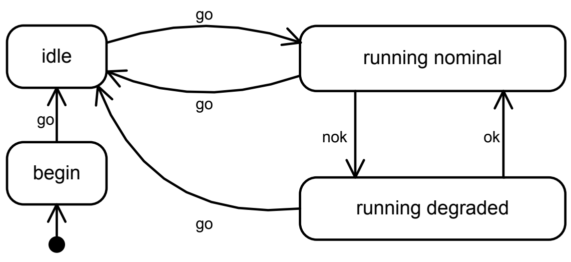

Figure 8: Flatten FSM without actions

Note that the actions are lacking. The super-state running disappeared and its incoming and outgoing transitions have been dispatched over its substates. These substates (nominal and degraded) are renamed according to their former encompassing state (running nominal and running degraded).

Contextualization. To generate the contextualization, we first need to *invert* the initial Modif specification using a dedicated tool written in ATL. The reverse Modif specification we obtain can take into account the original metamodel annotated with keys. We use it to generate the program corresponding to contextualization.

Then we apply it on the flat FSM depicted by figure 8 together with the original FSM of figure 7 annotated with keys. We obtain the flat FSM involving actions depicted by figure 9.

The transitions and the states that already existed in the original FSM are associated again to their original actions. The entry action run1 is associated to its original state (nominal) which is now running nominal. And because super-state running is removed, the entry and exit actions that are associated to it are lost.

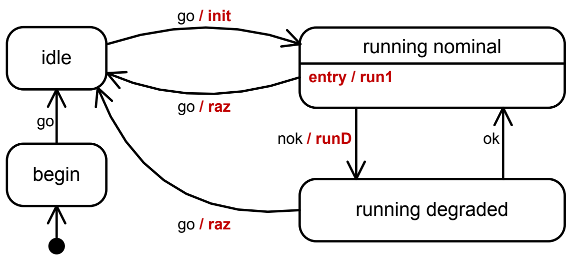

Figure 9: Contextualized flatten FSM

Lessons from this example. This simple example shows how we recover data that had been deleted before the application of a legacy tool. As for the actions that where associated to the deleted super-state, they cannot be *automatically* recovered. And if the flattener had performed creations of new instances instead of updating the existing ones, we would not have recovered any deleted actions.

Moreover, these tools are *generic* and they do not involve semantics issues. The instances they produce must (or may) be *completed* by other means. In the example, the *entry* and *exit* actions of state running have to be added using dedicated user code.

This work highlights the difficulties of reusing a legacy tool, especially when contextualization is needed. The way the legacy tool is designed impacts its reusability. It is higher for rewriting tools that give priority to *update* instead of *recreation*. In this context, the tools we propose are *helpers* made for designers in order lower the cost of rewriting functions that cannot be automatically reused by generative means.

4. RELATED WORKS

This work is on the border of several related fields: refactoring, model versioning, data migration and bidirectional transformations. It also adresses a well-known problem in the context of databases called the *view-update* problem [10]: under which conditions updates on views can be translated into updates on the underlying database? The constant complement translator approach for this problem involves a similar principle of our complementary set of initial instances annotated with keys. The initial context can be seen as the database and the tool's definition domain plays the role of a view. The tool we aim at reusing corresponds to the set of updates applied on the view. Our concern differs on two points. Firstly, some refactoring operations offered by Modif are out of the scope of the functions used to build views. For instance, we are working out the ability to extend an existing metamodel with Modif, and thus to add data at the instance level. Secondly, we do no make any assumption about how the legacy tool performs update. For now, we only consider its *effects*. These effects cannot be simply translated into a composition of *reversible* predefined updates.

The round-trip transformation we obtain from the composition of injection and contextualization implements a specific kind of *lense i.e.* a bidirectional transformation to propagate updates between connected structures [7, 2]. But maintaining the consistency of the initial context and the tool's definition domain was not our priority concern. We only wanted to transform instances without loss, even in the case of deletion. Thus our aim is much less ambitious than what is expected from lenses, like in the context of model versioning for instance. We focused on what can be recovered after the call of a black-box legacy rewriting tool.

The transformations specified by Modif are inspired by some works about refactoring [12], but they are applied at a metamodeling level. They notably aim at promoting co-evolution of models and metamodels [5, 6, 19]. Compared to these works, Modif introduces original operations (*e.g. hiding* of inheritance) and produces target metamodels. Modif is designed as a helper and allows to capitalize on repetitive operations related to co-evolution. From a higher point of view, injection is a generated data migration tool [17]. The novelty of our work is the contextualization. Indeed, for each generated injection, we are able to generate the reverse migration which enables to recover as far as possible data that had been removed. A similar approach to enforce the reuse of legacy transformations has been introduced in [11]. The idea is to rewrite legacy transformations in specific context using metamodel differences. But this approach is not automated and it does not consider the contextualization problem on which this paper is focused. The *domain evolution framework* defined in [18] also offers model versioning facilities focused on the evolution of a modeling language. Modif is a DSL made to target more precisely tool integration.

5. CONCLUSION

In order to complete the tool support of a DSML, we introduced a language based on a metamodel called Modif. It aims at promoting the reusability of legacy transformations. It can be seen as a tranformation language dedicated to semantically close metamodels. From a Modif specification, we can generate the data migration components. In this paper, we introduced a relational mechanism upon Modif to enable the reuse of endogenous transformations like rewriting tools. Now, we investigate their formal foundations and some extensions.

Graph semantics. We are working out a denotational semantics of Modif based on graphs to formally prove the soundness of the joint algorithm. This semantics is closed to KM3 [8] except it introduces within a same logical framework instances, metamodels and Modif specifications. This semantics is also intended to be used to prove that contextualization has no impact on what is performed by the legacy tool.

CRUD creation. To complete the basic operations Modif offer, we are adding the ability to *create* new data. We still intend to generate the injection with this new operation. For that purpose, we need to define an *object factory* mechanism based on customizable rules.

Levenshtein distance between two metamodels. The previous point is a preliminary work for a more ambitious issue. For now, a target metamodel can be generated from a source metamodel *plus* a Modif specification. We aim at generating a Modif specification from the given source and target metamodels. This generated Modif specification would come from the smallest *Levenshtein distance* between two metamodels. This distance is a metric for the difference between two sequences with regard to the basic edition operations that are *add*, *delete* and *replace*. As we mentioned before, if there is a need to build bridges between two metamodels, we can properly suppose they are semantically close. Then we expect the smallest distance between them is relevant.

6. REFERENCES

[1] J.-P. Babau and M. Kerboeuf. Domain Specific Language Modeling Facilities. In *proceedings of the 5th MoDELS workshop on Models and Evolution*, 2011.

[2] K. Czarnecki et al. Bidirectional transformations: A Cross-Discipline Perspective GRACE meeting notes, state of the art, and outlook.

[3] Eclipse Modeling Framework. http://www.eclipse.org/modeling/emf.

[4] W. Frakes and C. Fox. Sixteen questions about software reuse. *Communications of the ACM*, 1995.

[5] K. Garcés, F. Jouault, P. Cointe, and J. Bézivin. Managing model adaptation by precise detection of metamodel changes. In *Proceedings of ECMDA-FA*, 2009.

[6] M. Herrmannsdoerfer, S. Benz, and E. Juergens. COPE - automating coupled evolution of metamodels and models. In *Proceedings of ECOOP*, 2009.

[7] M. Hofmann, B. Pierce, and D. Wagner. Symmetric lenses. In *Proceedings of the 38th symposium on Principles of programming languages*, 2011.

[8] F. Jouault, J. Bézivin, and A. Team. Km3: a dsl for metamodel specification. In *In proc. of 8th FMOODS, LNCS 4037*, pages 171–185. Springer, 2006.

[9] F. Jouault and I. Kurtev. Transforming models with atl. In *MoDELS Satellite Events*, pages 128–138, 2005.

[10] J. Lechtenbörger. The impact of the constant complement approach towards view updating. In *Proceedings of the 22th symposium on Principles of database systems*, 2003.

[11] D. Mendez, A. Etien, A. Muller, and R. Casallas. Towards Transformation Migration After Metamodel Evolution. In *Model and Evolution Wokshop*, 2010.

[12] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 2004.

[13] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. Weaving executability into object-oriented meta-languages. In *Proceedings of MODELS/UML*, 2005.

[14] Object Management Group. *UML Infrastructure specification,* `http://www.omg.org/spec/UML`. 2007.

[15] OMG. *MOF QVT Final Adopted Specification.* Object Modeling Group, June 2005.

[16] Platypus. Technical Summary and download. `http//cassoulet.univ-brest.fr/mme`.

[17] L. M. Rose et al. A comparison of model migration tools. In *Proceedings of MODELS*, 2010.

[18] J. Sprinkle. *Metamodel driven model migration.* PhD thesis, 2003.

[19] G. Wachsmuth. Metamodel adaptation and model co-adaptation. In *Proceedings of ECOOP*, 2007.