Adaptation of Models to Evolving Metamodels

Kelly Garces, Frédéric Jouault, Pierre Cointe, Jean Bézivin

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

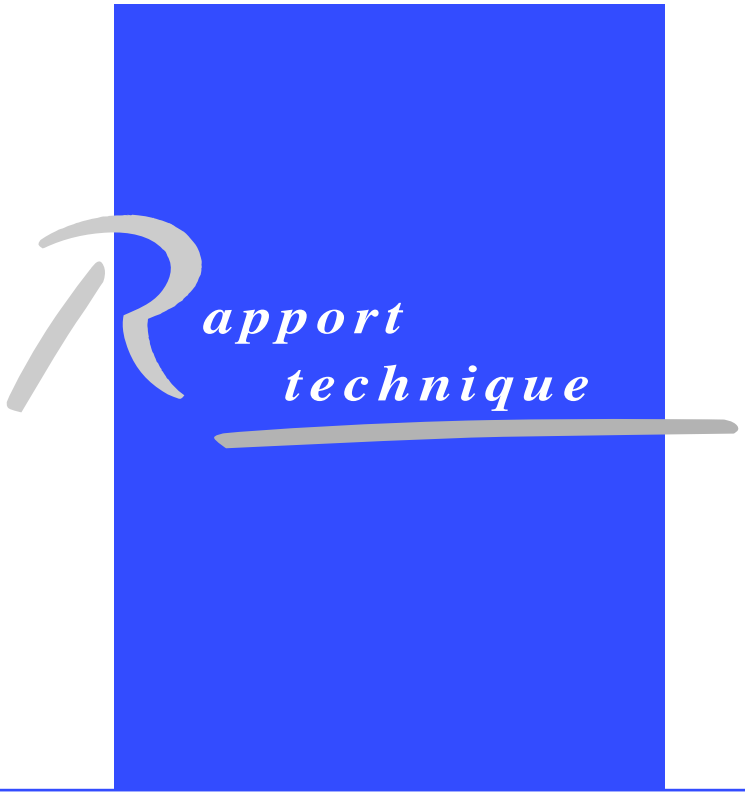
Adaptation of Models to Evolving Metamodels

Kelly Garcés — Frédéric Jouault — Pierre Cointe — Jean Bézivin

N° 6723

November 2008

Thème COM



ISSN 0249-0803 ISRN INRIA/RT--6723--FR+ENG

Adaptation of Models to Evolving Metamodels

Kelly Garcés [*] [†] , Frédéric Jouault [*] , Pierre Cointe [†] , Jean Bézivin [*]

Thème COM — Systèmes communicants
Projets Obasco et AtlanMod

Rapport technique n° 6723 — November 2008 — 27 pages

Abstract: The problem of automatic or semi-automatic adaptation of models to their evolving metamodels is gaining importance in the Model-Driven community. Recent approaches propose to adapt models using predefined information (i.e., a trace of changes). Unfortunately, this information is not always available in practice. In many situations metamodels evolve without keeping track of the applied changes. We propose a more general two step solution. First step computes equivalences and differences between the metamodels and saves these into a "weaving model". This weaving model acts as a high-level specification of adaptation transformation. Second step translates this model into an executable transformation. This technical report shows the results obtained in applying the approach on two concrete scenarios: a Petri net metamodel, and the Netbeans Java metamodel.

Key-words: Model-Driven Engineering, Model Transformation, Adaptation.

[*] AtlanMod team
[†] Obasco project team

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex (France)
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Adaptation des Modèles à l'évolution de leurs Métamodèles

Résumé : La problématique de l'adaptation automatique ou semi-automatique de modèles en fonction des évolutions de leurs métamodèles est centrale dans la communauté de l'ingénierie de modèles. Les approches courantes proposent d'utiliser l'historique des changements effectués sur les métamodèles pour faire évoluer les modèles en conséquence. Cependant en pratique, cette information n'est pas toujours disponible. Bien souvent, les métamodèles évoluent sans possibilité de tracer l'historique des modifications appliquées. Nous proposons une solution générale qui se décompose en deux phases. La première phase consiste à calculer les équivalences et les différences entre les métamodèles pour les conserver au sein d'un "modèle de tissage". Nous considérons un modèle de tissage comme une spécification abstraite d'une transformation d'adaptation. La deuxième phase consiste à traduire ce modèle en une transformation exécutable. Ce rapport montre les premiers résultats significatifs obtenus en appliquant notre approche sur deux cas concrets: un métamodèle de Petri Net et le métamodèle de Netbeans Java.

Mots-clés : Ingénierie Dirigée par les Modèles, Transformation de Modèles, Adaptation.

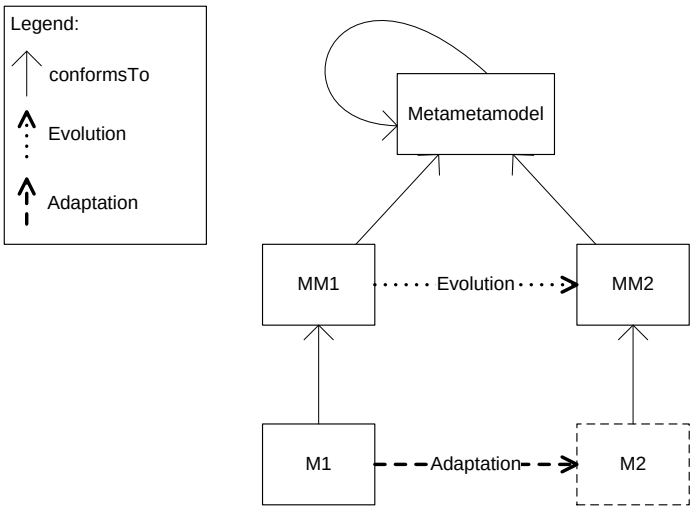


Figure 1: Metamodel evolution and model adaptation

1 Introduction

The adaptation of software systems is a rapidly increasing need due to influence of technological and business innovations, changes in legislation and continuing internationalisation [1].

The Model-Driven (MD) software systems are not beyond that need. An MD system basically consists of metamodels, terminal models, and transformations. A metamodel is composed of concepts and relationships. A terminal model contains instances of the metamodel concepts. A transformation can translate the instances conforming to a metamodel concept into instances conforming to another concept.

Because every model conforms to a metamodel, changes in a metamodels may impact many other artifacts: 1) models that conform to it, 2) transformations that take it as source and/or target, 3) syntax specifications (i.e., textual, or visual), etc. Therefore, the adaptation to metamodel evolution is a relatively broad problem. In this work, we focus on 1), which consists in adapting models so that they conform to a new version of their metamodel.

Fig. 1 gives an overview of metamodel evolution and its associated model adaptation. During the software project lifecycle, a metamodel *MM1* may evolve into a metamodel *MM2* to fix bugs and/or support extra features. *MM1* and *MM2* are represented at the level of metamodel. They both conform to the same metametamodel[1]. Evolution is represented on the figure by a dotted arrow between the two metamodels. When this happens, it is necessary to adapt the terminal model *M1* (conforming to *MM1*) to the new metamodel *MM2*. The objective of adaptation is to generate a terminal model *M2* from a terminal model *M1*. Adaptation is represented by a dashed arrow between the two terminal models. The dashed outline of *M2* illustrates that this model is the output of the adaptation process, whereas all other models pre-existent.

[1]Our definition of Metamodel, metametamodel, and terminal model are precisely defined in [2].

Metamodel evolution is usually performed manually by a group of engineers. Model adaptation can be done directly by a developer, or indirectly by creating a program. Direct creation by a developer may require a significant amount of time if many models have to be created, and it can be an error-prone task. Creation using a program is only partially automatic as the developer has to write the code. Let us suppose a developer adapts the models by implementing model transformations. Before writing the transformations, the developer should discover the equivalences [2] (i.e., pairs of equal of at least similar elements in two metamodels) and changes (additions, deletions, modifications). Subsequently, s/he may reuse pre-existent code to deal with those equivalences and changes. As a result, s/he spends a lot of time, and gets an ad-hoc (new programs are required for each scenario) and error-prone (because of the copy and paste action) solution. How can we automatize the adaptation transformation development ?

We propose a two step solution that semi-automatically generates adaptation transformations. Firstly, a matching process computes the equivalences and changes between *MM1* and *MM2* by executing a set of heuristics. Secondly, an adaptation transformation is derived from the found equivalences and changes.

The first step is an instance of the diff problem, which has been thoroughly studied in many contexts. For example, the Unix diff compares two text files [3]. It reports a minimal list of line-based changes. This list may be used to bring either file into agreement with the other (i.e., patching the files). Another example is the diff between two Matlab/Simulink diagrams [4]. Related tools (e.g., SiDiff [5]) often represent the diagrams as graphs. The differences between the graphs are computed according to their structure. The tools may display the differences by using different colors. In our case, we compare two metamodels (i.e., a given metamodel to a former version of the same metamodel), and the objective is to bring older terminal models into agreement with the newer metamodel. The comparison discovers equivalences, as well as simple and complex changes between the two metamodels. A simple change describes the addition, deletion and/or update of one metamodel concept. A complex change integrates a set of simple changes affecting multiple concepts. The comparison result is used in a second step to derive an adaptation transformation.

We evaluate the performance and precision of our approach by applying it on two case studies: a Petri net metamodel from the research literature, and the Netbeans Java metamodel. We investigate 6 versions of the Petri net metamodel (containing between 10 and 20 elements), and 8 versions of the Java metamodel (containing approximately 250 elements). Using this approach, we are able to analyze, on a desktop machine, any pair of Petri net metamodels in under 1 second, and any pair of Java metamodels in under 10 seconds. Moreover, our prototype always discovers the changes, and only fails to identify changes when in truth there is an equivalence (1% of the cases). We have implemented the prototype on the Atlas Model Management Architecture (AMMA) platform [6].

This paper is organized as follows: Section 2 compares our contributions to other known solutions. Section 3 presents two scenarios illustrating the whole approach. Section 4 uses these examples to describe how the metamodels may change. Section 5 details the model adaptation problem. Section 6 presents an approach handling this problem. Section 7 describes the first significant results

[2] The equivalences are commonly called mappings or correspondences. We indistinctly use these terms.

obtained by applying our approach on concrete scenarios. Section 8 concludes the technical report.

2 Related works

We may divide the related approaches according to which of the two main issues they deal with: 1) discovery of equivalences and differences, or 2) derivation of adaptation transformations.

As we pointed out in the introduction, the diff problem has been tackled in several domains. We now describe the closer related works. In the context of relational and object-oriented data bases, the production of equivalences between two schemas/ontologies has been studied in [7, 8]. In the MDE domain, the approaches in [9, 10, 11, 12] present algorithms for detecting changes between UML models. Sriplakich at al. [13] identify simple changes in terminal models conforming to any metamodel. Wenzel and Kelter [14] present an approach which discovers fine-grained traces between versions of modeling languages, e.g., UML models, schemas, Web service description languages, and domain specific languages. Finally, the EMF Compare tool [15] reports simple changes between terminal model pairs or metamodel pairs.

In contrast to the first issue, the second one has been addressed by only a few recent approaches. Wachsmuth [16], Gruschko et. al [17], and Cicchetti et al. [18] assume the traces of changes are available and derive adaptation transformation from those changes. In particular, Cicchetti et al. obtain the changes by using available tools (including EMF Compare).

The following four items position our approach in comparison with the previously cited solutions:

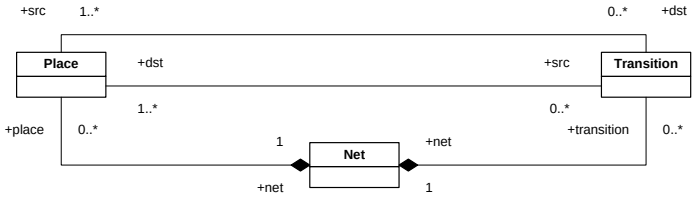
1. Similarly to [14], our approach computes equivalences and differences between any pair of metamodels (e.g., representing schemas, UML models, ontologies) .

2. Our solution overlaps the solutions presented in [16, 18] in the sense of considering both simple and complex changes.

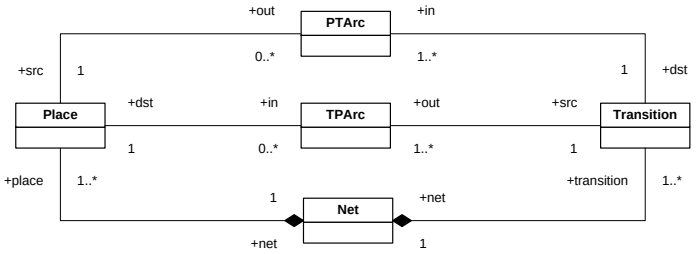
3. Unlike existing approaches [16, 17, 18], we do not suppose that the changes are already known. We consider a more general case where the evolution of metamodels is done without someone explicitly keeping track of the applied changes.

4. The matching step executes modularized heuristics that discover the differences between the metamodels. While most of the previous approaches (with the exception of [8]) execute all the heuristics, each of our heuristics may be plugged or unplugged on demand, which may mean a considerable performance increase.

To sum up, most of the listed works solve the two main issues in an isolated fashion. Some of them are in contexts different to the metamodel evolution. In contrast, we propose a solution that addresses all the described model adaptation issues in a consistent and integrated way.



(a) Petri Net *MM1*



(b) Petri Net *MM2*

Figure 2: Petri Net metamodels

3 Two motivating scenarios

This section presents in detail the scenarios illustrating the paper. The Petri Net metamodel is selected because it is small, well known, and includes interesting changes. The concrete choice of Netbeans Java is driven by two main reasons: experimental data is widely available (open-source), the metamodel and models are significantly larger than those of Petri Net, and therefore illustrates the scalability of our approach.

3.1 A Sample Petri Net Metamodel

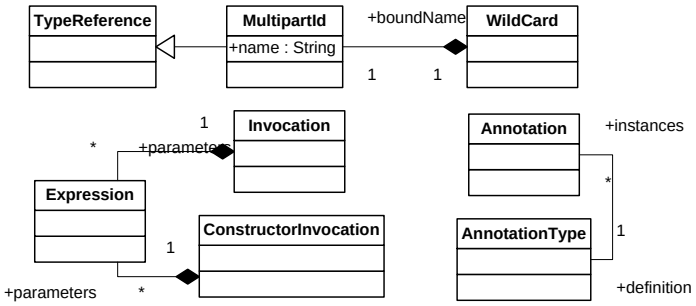
This scenario is based on the six versions of the Petri Net metamodels provided by [16]. Fig. 2 illustrates the versions 0 (*MM1*) and 2 (*MM2*). *MM1* represents simple Petri Nets. These nets may consist of any number of places and transitions. Such transition has at least one input and one output place. *MM2* represents more complex Petri Nets. The principal changes between *MM1* and *MM2* illustrated in Fig. 2 are:

- `place` and `transition` references change its multiplicity from 0-* to 1-*.
- `PTArc` and `TPArc` classes as well as `src`, `dst`, `in`, `out` references are added.

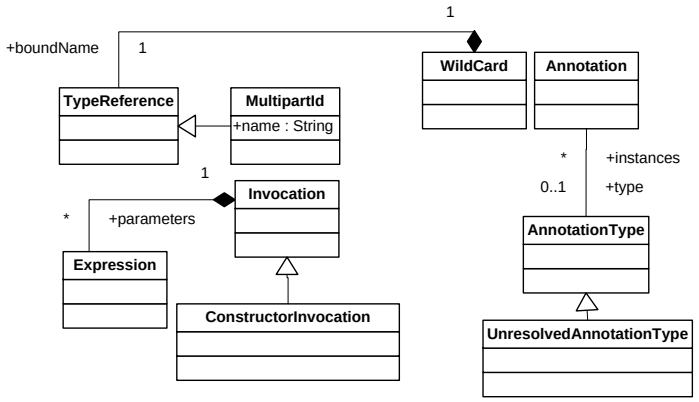
In our experimentation, we create terminal models conforming to *MM1* from the Petri Net models described in [16].

3.2 The Netbeans Java Metamodel

This scenario is based on the evolution of the Netbeans Java metamodel [19]. The Java tooling of this opensource IDE is built around an explicit MOF 1.4 metamodel. Netbeans provides more than thirty versions (going from version



(a) Java *MM1*



(b) Java *MM2*

Figure 3: Netbeans Java metamodels

1.1 to version 1.19 plus several branches) of the Java metamodel available via a public versioning system (namely CVS). Fig. 3 illustrates two versions of the Netbeans Java metamodel: version 1.12 (3(a)), and version 1.15 (3(b)). *MM2* includes the following changes:

- `UnresolvedAnnotationType` class is introduced.
- `boundName` reference is generalized. In the newer metamodel, `boundName` associates `WildCard` and `TypeReference`. `TypeReference` is a superclass of `MultipartId`.
- `definition` reference between `Annotation` and `AnnotationType` is generalized in terms of its multiplicity, this changes from 1-1 to 0-1. Additionally the name of the reference changes from `definition` to `type`.
- `ConstructorInvocation` class gets `Invocation` superclass.

In addition to the metamodels, it is also necessary to get terminal models conforming to *MM1*. We parse Java programs using the Netbeans API to obtain the terminal models. Fig. 4 shows how *M1* is obtained from a Java Program by using *Java Parser 1*. The parser and metamodel versions have to match. Because there is a version of the parser for each version of the metamodel, a possible way to obtain a version of the model conforming to a new metamodel

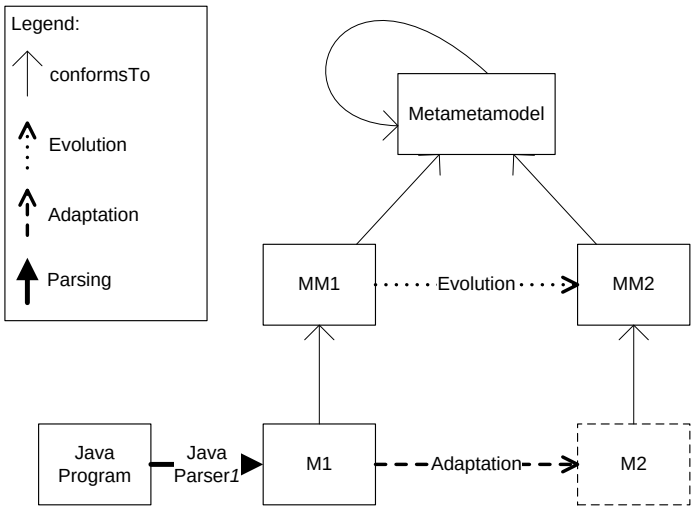


Figure 4: Overview of Java scenario

is to reparse the Java code with the corresponding parser [3]. In our experimentation, we get *M1* from a Java program.

4 A Classification of Metamodel Changes

B. Gruschko et al. [17] classify the metamodel changes into three categories:

- **Non breaking changes.** Changes occurring in the metamodel that do not impact the models conforming to this metamodel.
- **Breaking and resolvable changes.** Changes impacting the models that can be resolved automatically.
- **Breaking and irresolvable changes.** Changes impacting the models that cannot be resolved automatically.

No adaptation is needed for the changes from the first category. Our approach aims to adapt models to the second category changes. The third category requires user assistance, this category is out of scope of this report. We illustrate below the three categories using concrete metamodel changes. The classification presented here is an initial work containing the metamodel changes identified in [16], and our experimentations. If *breaking and resolvable changes* category is further refined then some cases belonging to *non breaking changes* may appear. For example, *Eliminate class* change could belong to *non breaking changes* if the eliminated class is abstract.

4.1 Non Breaking Changes

This category contains the following changes:

[3]The reader interested in the generation of a parser from a couple *(grammar, metamodel)*. may consult the Eclipse projects Modisco [20], and TCS [21] [22].

- **Introduce class.** A new class is introduced. For instance, `UnresolvedAnnotationType` is a new class of the Java *MM2*.
- **Introduce property.** An optional property (i.e., not requiring a default value) is added.
- **Generalize property.** A property is generalized in terms of its multiplicity and/or of its type. A first example in the Java scenario is the generalization of reference `boundName` between `WildCard` and `MultipartId`. In the new metamodel, `boundName` is between `WildCard` and `TypeReference`, which is a superclass of `MultipartId`. Additionally, `definition` reference between `Annotation` and `AnnotationType` is generalized in terms of its multiplicity, which changes from 1-1 to 0-1.
- **Push property.** A property is eliminated from a superclass and introduced into each subclass.
- **Extract superclass.** A set of properties common to a set of classes is extracted into a new superclass.

4.2 Breaking and Resolvable Changes

The category contains the changes:

- **Introduce restricted property.** A new mandatory property (i.e., requiring a default value) is introduced.
- **Eliminate class.** A class is eliminated from a metamodel.
- **Eliminate property.** A property is eliminated from a class.
- **Pull property.** A property is pulled into a superclass.
- **Rename element.** An element is renamed. For instance, the reference between `Annotation` and `AnnotationType` is renamed from `definition` to `type`.
- **Move property.** A property is moved along a one-to-one relation. This implies the elimination of the property from the source class and the introduction it in the target class.
- **Extract/inline class.** A set of properties is extracted from a class to a new introduced class. This requires a one-to-one relation between the new container class and the affected class. Then the properties are moved along this relation into the new class. *Class inlining* is the inverse operation. In the Petri Net example the references between `Place` and `Transition` are extracted into `PTArc` and `TPArc` classes.
- **Flatten hierarchy** A superclass is eliminated and all of its properties are pushed into the subclasses.
- **Introduce/Delete superclass.** A class gets another class as its superclass. `ConstructorInvocation` class gets `Invocation` superclass. *Delete superclass* means to eliminate the inheritance relationship between a class and one of its superclasses.

4.3 Breaking and Irresolvable Changes

The category consists of:

- **Restrict property.** A property can be restricted in terms of its type and/or its multiplicity. A type restriction example is the change from String to Integer. This change can produce invalid conversions. Restricting the type of a property implies the conversion of the type of each value. A multiplicity restriction example is illustrated by the Petri Net scenario; `place` reference between `Place` and `Net` changes its multiplicity from 0-* to 1-*. The Petri Net *M1* may become invalid because `Net` must now comprise at least one `Place`. Restricting the lower bound requires new values for the property, these values should be usually provided by the user. Finally, restricting the upper bound of the multiplicity may require a selection of certain values.

5 Problem description

Metamodel evolution requires model adaptation, i.e, to transform *M1* models into *M2* models. Section 1 presents some ways to do that, but we focus on the adaptation using model transformations. The development of adaptation transformations implies two tasks: 1) Discover the equivalences and changes between the metamodels, and 2) Apply transformation patterns to deal with those equivalences and changes. For example, in the Java scenario, the developer should note an equivalence between `Annotation` of *MM1* and `Annotation` of *MM2*, and a change in `definition` which is renamed to `type`. In the Petri Net scenario, the developer should discover an equivalence between `Place` of *MM1* and `Place` of *MM2*, and a change in `dst` which is extracted into `PTArc`. After discovering equivalences and changes, the developer has to implement the model transformations using a concrete model transformation language. She should implement transformation rules which consist of source patterns (which match classes of *MM1*), target patterns (which create classes of *MM2*) and bindings (which initialize properties of *MM2* using properties of *MM1*). Listings 1 and 2 illustrate the transformation rules for our scenarios using the Atlas Transformation Language (ATL)[4].

Listing 1: Transformation excerpt for the Java scenario

```
rule Annotation2Annotation {
  from
    a1 : MM1!Annotation
  to
    a2 : MM2!Annotation (
      type <- a1.definition,
      name <- a1.name
    )
}
```

Listing 2: Transformation excerpt for the Petri Net scenario

```
rule Place2Place {
  from
    pv1 : MM1!Place
  to
    pV2 : MM2!Place (
```

[4]ATL is a transformation language based on declarative and imperative rules [23].

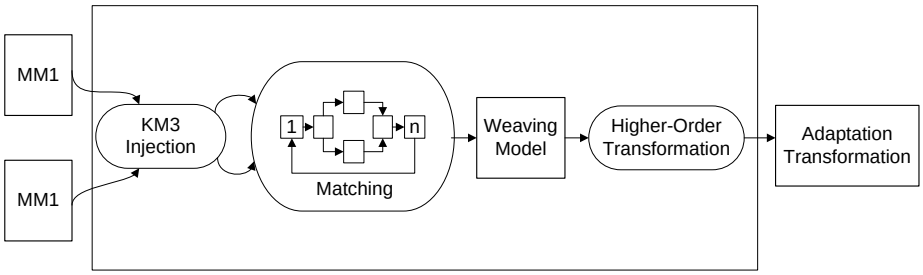


Figure 5: Approach Overview

```
6         out <- pv1.dst -> collect (tV1 | thisModule.dstPTArc(tv1, pv1)}))
7       )
8 }
9
10 unique lazy rule dstPTArc {
11   from
12     transition : MM1!Transition,
13     place : MM1!Place
14   to
15     tV2 : MM2!PTArc (
16     dst <- transition
17   )
18 }
```

In Listing 1, `Annotation2Annotation` rule matches/creates elements conforming to `Annotation` (lines 3 and 5) and initializes `type` using the values of `definition` (line 6). In Listing 2, `Place2Place` rule matches/creates elements conforming to `Place` (lines 3 and 5) and initialize `dst` of `PTArc` (lines 18) using the values `dst` of `Place` (line 6).

The discovery task may be expensive if the metamodels are quite large or do not include evident changes. Moreover, the developer may spend a lot of time implementing the transformation rules from the scratch. On the other hand, if s/he copies and pastes pre-existent code describing similar equivalences or changes, s/he likely to make mistakes. How can we automatize both discovery of equivalences/changes and implementation of transformations ?

6 Our approach

We propose an approach that semi-automatically generates adaptation transformations (see Fig. 5). Basically, both metamodels *MM1* and *MM2* are translated into the KM3 notation. A *matching strategy* discovers equivalences and changes between the metamodels. Equivalences and changes are represented by a matching model. A HOT takes this model as input to generate a model transformation as output. This transformation converts a *M1* model (conforming to *MM1*) into a *M2* model (conforming to *MM2*). The subsections below present our approach in detail.

6.1 The Kernel Metametamodel (KM3)

The metamodels in KM3 notation are represented as graphs [22]. A graph consists of nodes and edges. The nodes denote elements conforming to the KM3

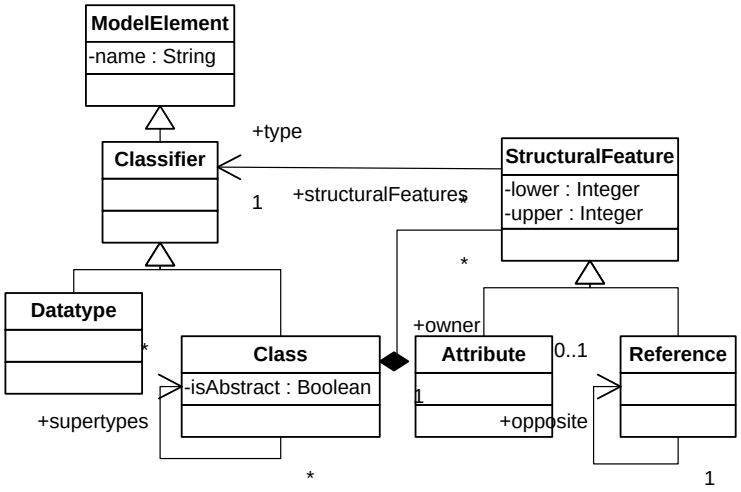


Figure 6: KM3 concepts

concepts. The edges represent relationships (i.e., inheritance, containment, and association) between two nodes.

Fig. 6 shows the basic concepts of KM3. The `ModelElement` class denotes concepts that have a `name`. `Classifier` extends `ModelElement`. `DataType` and `Class` in turn specialize `Classifier`. `Class` consists of a set of `StructuralFeatures`. There are two kinds of structural features: `Attribute` or `Reference`. `StructuralFeature` has `type` and `multiplicity` (`lower` and `upper` bound). `Reference` has `opposite` which enables to get the owner and target of one reference. `Class` may extend zero or more other classes and may be abstract.

In the Java metamodel, `Annotation` is a node conforming to the `Class` concept. Other node is `instances` which conforms to the `Reference` concept. The node `instances` has the attributes `lower` and `upper` with value `0` and `*`. Finally, an edge example, is the edge representing a containment relationship between the `Annotation` class and the `instances` reference.

6.2 Matching model

The equivalences and changes between the metamodels are represented by a matching model. This model conforms to a *Matching metamodel*[5] illustrated by Listing 3. The main concept is `Equal` which describes a mapping between an element of *MM1* (`leftElement`) and an element of *MM2* (`rightElemnt`). `Equal` has a similarity value (between 0 and 1) that represents the plausibility of the correspondence. `Equal` can be extended to describe more specific equivalences. For example, its extensions can represent simple equivalences (e.g., `EqualClass`, `EqualReference`, `EqualAttribute`) or modifications (e.g., `Renamed`). Other basic concepts are `Added` and `Deleted` which mark a metamodel element as deleted/added from/into *MM1*.

Listing 3: Excerpt of a matching metamodel

```
1  class LeftElement extends WLinkEnd {}
```

[5]This metamodel extends the core weaving metamodel proposed by [24].

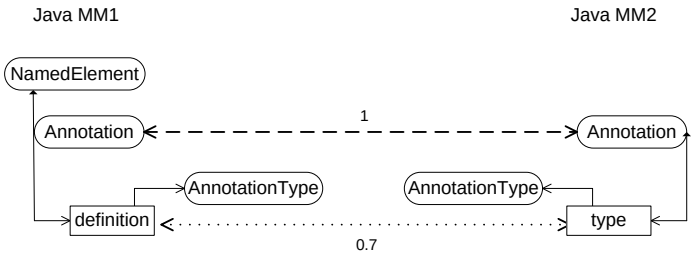


Figure 7: Graphical notation of a matching model

```
2
3   class RightElement extends WLinkEnd {}
4
5   class Link extends WLink {
6     reference left[0-1] container : LeftElement;
7     reference right[0-1] container : RightElement;
8   }
9
10  class Equal extends Link {
11    attribute similarity : Double;
12  }
13
14  class EqualClass extends Equal {}
15
16  class EqualAttribute extends Equal{}
17
18  class EqualReference extends Equal{}
19
20  class Renamed extends Equal{}
21
22  class Added extends Link {}
23
24  class Deleted extends Link {}
```

We can represent an instance of `Equal` using a graphical [6] or a textual notation. Fig. 7 shows a graphical notation for the Java scenario. The elements of Java *MM1* are located on the left-hand side. The elements of Java *MM2* are located on the right-hand side. The ovals represent metamodel classes. The boxes represent attributes and references. The solid arrows represent relationships between metamodel elements (containment, inheritance, association). The dashed/dotted arrow represents instances of `Equal`. Additionally, they have a number representing the similarity value (e.g., 0,7). In particular, the dashed arrows represent instances of `EqualClass` and the dotted arrows represent instances of `EqualReference`. The dashed and dotted arrows link the classes `Annotation` and `definition` and `type` references indicating that they are equivalent.

An instance of `Equal` can be also textually represented as the pair `(a, b)`, where `a` references an element of *MM1* and `b` references an element of *MM2*. The pair `(Annotation, Annotation)` represents the dashed arrow. The instances of `Equal` are created, modified, and deleted by matching strategies.

6.3 Matching strategy

Given two metamodels in KM3 notation, *MM1* and *MM2*, a matching strategy discovers the equivalences and changes between them by executing a set

[6] This is an adaptation of a graphical notation for ontologies proposed by [25].

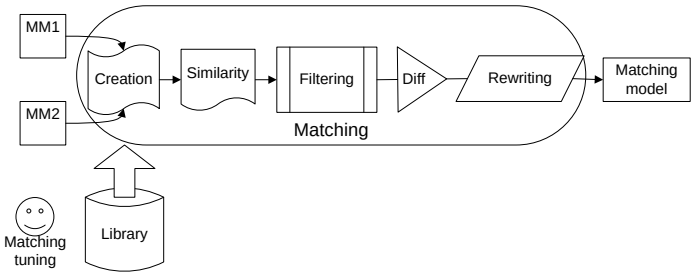


Figure 8: Matching Strategy Overview

of heuristics. Fig. 8 is a intuitive (there may be more elaborated matching strategies, e.g., including loops) overview of what a matching strategy is. The figure presents the kinds of heuristics and their execution order. At first a *creation* heuristic prepares a collection of equivalences to be use by the subsequent heuristics. Afterward, the *similarity* heuristics compute a similarity value for each equivalence based on the names, internal properties, or structures of the matched elements (`leftElement` and `rightElement`). The *name* similarity heuristics are first executed and then the other similarity heuristics (*multiplicity*, *context*, *internal properties*, and *context*). That is because the name is a "safe" indicator that left and right are the same [10]. In particular, the context heuristic strengthens the previously computed similarity values by comparing the relationships of the matched elements. The *filtering* heuristics select equivalences taking into account the confidence value computed by the similarity heuristics. The *differentiation* heuristic recognizes equivalences and changes. The matching step finishes when the *rewriting* heuristics reorganize the matching model to make it more similar to adaptation transformations. Moreover, the user can refine the heuristics outputs.

Let us now conceptualize the kinds of heuristics briefly presented in the previous paragraph. Each kind of heuristic describes particular implementations.

6.3.1 Creation

The creation heuristics create mappings referencing metamodels elements. A new mapping is created only if the metamodel elements match a specific criterion. A simple creation heuristic is *Creation by type*. This creates a mapping between two metamodel elements if they have the same type (i.e., classes, attributes, or references). Another *Creation* heuristic, named *Creation by Type and FullName*, creates mappings between every pair of elements with the same type and full-name. In the Java scenario, *Creation by type* creates an equivalence between `Annotation` of *MM1* and `Expression` of *MM2* because these are classes.

6.3.2 Similarity

The similarity heuristics compute a similarity value for each mapping by comparing properties (e.g., name, multiplicity, context) of the matched metamodel elements. A high similarity value indicates that there is a good probability of

equivalence between the matched metamodel elements. We describe below four more specific similarity heuristics.

- **Name Similarity** compares the names of metamodel elements (classes, attributes, references) in different ways:
  - *String similarity* compares names using string comparison algorithms such as Levenshtein, n-grams, and edit distance [26].
  - *Dictionary of synonyms* compares names using a dictionary of synonyms. We use WordNet [27] for this purpose. This dictionary organizes nouns, verbs, adjectives, and adverbs into synonym sets. We consider that two elements are similar if they are in the same synonym set.
- **Multiplicity Similarity** compares the multiplicity of references and attributes. It assigns a similarity value to mappings connecting metamodel references that have the same multiplicity (lower and upper bounds). When this heuristic is applied on the Petri Net scenario, it assigns to `(net, net)` a similarity value equal to 1. That is because `net` is a reference between `Place` and `Net` whose multiplicity does not change. In contrast, the similarity value of `(place,place)` mapping is not modified because the reference `place` between `Net` and `Place` changes its multiplicity from (0..*) to (1..*).
- **Context Similarity** compares the relationships of the metamodel elements. For example, it compares attributes/references contained by a given class, its superclasses, and its associated classes. Our implementation of the context similarity transformation is inspired from the *Similarity Flooding* algorithm (SF) [28]. It propagates the similarity values between elements assuming that these are similar when their adjacent elements are similar. The algorithm is executed in two steps. The first step associates two mappings (*l1* and *l2*) if there is a semantic relationship between them. The second step propagates the similarity value from *l1* to *l2* because of the association. In the Java scenario, the first step associates `(Annotation, Annotation)` and `(definition, type)` because `Annotation` of *MM1* contains `definition` reference and `Annotation` of *MM2* contains `type` reference. Then, the second step propagates the similarity value from `(Annotation, Annotation)` to `(definition, type)`.
- **Similarity by Internal Properties** aggregates the similarity values computed by the previous heuristics (e.g., name, multiplicity). Every similarity value is a relative value which is multiplied by a weight. These amounts are summed to calculate a net similarity value.

6.3.3 Filtering

The previous heuristics may have created unwanted equivalences (i.e., equivalences with low similarities). The filtering heuristics select the equivalences satisfying a set of conditions. A basic filtering heuristic is *Threshold*. This selects the mappings with a similarity value higher than a given threshold value.

6.3.4 Differentiation

This heuristic compares the KM3 models to the remaining `Equal` mappings to identify `Added`, and `Deleted` mappings. The differentiation heuristic is executed if the user asks for. The matching process does not use these `Added` and `Deleted` mappings to derive the adaptation transformation.

6.3.5 Rewriting

Before this step, most mappings are contained in a single large collection. This heuristic reorganizes/retypes the mappings in order to capture the transformation patterns of a specific model transformation language. In our experimentation, we implement the following rewriting heuristics to represent ATL transformation patterns:

- **Basic.** This moves `EqualAttribute` and `EqualReference` mappings into `EqualClass` ones. In an ATL transformation, the classes matched by `EqualClass` are taken as the source/target patterns, and the contained `EqualAttribute`/`EqualReference` as the bindings.
- **Flattening.** This heuristic is necessary to deal with multiple inheritance because ATL does not support it. This copies the `EqualAttribute` and `EqualReference` mappings contained in `EqualClass` (e.g., matching class C) into every `EqualClass` matching a class that inherits directly or indirectly from C. For example, `Annotation2Annotation` rule in Listing 1, contains (e.g., `name <- a1.name`) binding which initialize `name` attribute. This attribute does not belong to `Annotation` but to one of its parent classes.
- **Complex changes.** This heuristic changes the type of mappings from `Equal` to other types describing complex changes. For example, if the set of properties of *A* class (*MM1*) are extracted into *C* class (*MM2*), then *(A, C)* should be retyped `Extracted`. This type of mapping generates two ATL transformations similar to those illustrated by Listing 2.

6.4 Higher-Order Transformation (HOT)

A HOT takes as input the final matching model produced by the matching strategy and generates as output a model transformation written in a particular transformation language (e.g., ATL, XSLT, SQL-like). For instance, the matching model in Fig. 7 is translated into the transformation of Listing 1. The `(Annotation, Annotation)` mapping generates the `Annotation2Annotation` transformation rule (line 1). The (`definition`, `type`) mapping generates `type <- a1.definition` binding (line 6).

7 Experimentation

7.1 Prototype implementation

We validate the performance and precision of our approach by applying a prototype on the two scenarios. We implement the prototype on the AMMA platform [6]. More specifically, we use the Atlas Model Weaver (AMW) [29] to work with

Table 1: Metamodel elements

Example	PetriNet			Java		
Version	0	1	2	1.12	1.13	1.15
Elements	11	11	21	255	256	258

matching models, and we specify the heuristics and HOT in ATL. The HOT generates the adaptation transformation in ATL code. In particular, we develop a library that contains the heuristics described in Section 6.3.

7.2 Data set

We have results from early experimentations which use 8 versions of the Netbeans Java metamodel, 6 versions of a Petri Net metamodel provided by [16], and 8 heuristics implemented using ATL. Three versions of each metamodel are selected. These versions are chosen because they contain significant changes. In the Java scenario, we choose the versions 1.12, 1.13, and 1.15. In the Petri Net scenario, we use the versions 0, 1, and 2. We make couples with these versions. Table 1 shows the versions and their number of elements (classes, attributes and references).

7.3 Evaluation criteria

The experimentation results are evaluated according to two criteria: performance and precision. These criteria are proposed in [25] to evaluate ontology matching strategies. The first one assesses the resources consumed by our process [7]. The second criterion compares the expected changes to the probable changes.

7.4 Procedure

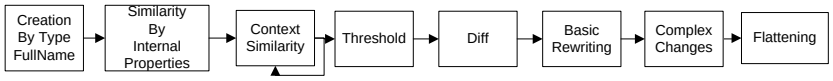
We apply three different matching strategies on the scenarios. First, we apply a basic matching strategy (see Fig. 9(a)) to both scenarios obtaining low performance and precision. In the Java scenario, *Creation by type* creates too many mappings. This amount of mappings diminishes the performance of the subsequent heuristics. In addition, the *Name Similarity* heuristic calculates low similarity values because this only takes into account the *name* property. As a consequence, the *Threshold* heuristic eliminates relevant mappings. On the other hand, in the Petri Net scenario, the yielded adaptation transformation does not generate the expected *M2* models because the matching models only represent simple model transformation patterns.

Because the basic matching strategy has low performance and precision, we develop more suitable matching strategies for each scenario. *Matching Strategy A* (Fig. 9(b)) matches the Petri Net metamodels (which are small and include complex changes, i.e., not only renamed elements), and *Matching Strategy B* (Fig. 9(c)) matches the Netbeans Java metamodels (which are big).

[7] The process is executed on a machine with Intel Core Duo processor and 1GB RAM.

(a) Basic Matching Strategy

(b) Matching Strategy A



(c) Matching Strategy B

Figure 9: Matching Strategies

7.5 Results

7.5.1 Performance

The time consumed by the matching process depends on the metamodels size. Table 2 shows the consumed time to match the couples of versions. In the Petri Net scenario, the matching process consumes less than 1 second. In the Java scenario, the matching process approximately takes 10 seconds. In the Java scenario, the *Context Similarity* heuristic consumes around 85% of total time because it is executed three times. If the *Similarity by Internal Properties* heuristic is refined (i.e., setting up the weights) then executions of *Context Similarity* heuristic may be saved. *Flattening* is not executed into the Petri Net scenario because this scenario does not have multiple inheritance. As this transformation is not run, execution time is saved in the Petri Net scenario. As a final point even if the matching step consumes a relevant amount of resources, we should keep in mind that this process generates an adaptation transformation that can be used several times.

7.5.2 Precision

We evaluate the precision of the matching strategies and the HOT.

Matching strategies. These are evaluated by comparing the expected changes (Section 4 describes some of them) to the probable changes (computed by the matching strategy and saved in a matching model). Table 3 displays the number of probable changes (modifications, deletions and additions) and some descriptions. In the Petri Net example (couple 0-2), the probable values matches the expected values. Thus, we get the modifications of type *Extract class* (e.g., `src` reference is extracted from `Place` to `PTArc`). The matching also detects the additions (e.g., `PTArc` and `TPArc`). No deletions are detected because there are no deletions between versions 0-2. In the Java example (couple 1.12 1.15), the matching step computes all the modifications of type *Introduce/Delete superclass* (e.g., `ConstructorInvocation` gets `Invocation`). A detected significant modification is the renaming of one reference from `definition` to `type`. All the additions (e.g., `UnresolvedAnnotationType`) are computed. The matching

Table 2: Time consumption

Scenario	PetriNet		Java	
Couple of version	0-1	0-2	1.12-1.13	1.12-1.15
Creation by Type and Full-Name	0.077	0.249	0.282	0.501
Similarity by Internal Properties	0.094	0.203	0.33	0.39
Context Similarity	0.157	0.14	8.16	8.73
Threshold	0.016	0	0.06	0.06
Complex changes	0.157	0.062	0.30	0.25
Diff and Basic Rewriting	0.062	0.281	0.16	0.19
Flattening			0.27	0.23
Total time (s)	0.563	0.935	9.548	10.36

strategy detects one false deletion (`parameters` of `ConstructorInvocation`) which actually corresponds to one modification. That is because *Creation by Time and FullName* does not create a mapping between parameters of `ConstructorInvocation` (*MM1*) and parameters of `Invocation` (*MM2*). As the mapping is missing, subsequent transformations do not discover the similarity between the elements. In general, the matching step precision depends on the intention of the users, s/he may want to adjust result manually until the intended equivalences [30]. We use the AMW tool to refine the computed mappings in a high abstraction level, the user is not supposed to go into the adaptation transformation (low abstraction level) generated from the mappings. On the other hand, we experiment different thresholds and weights to understand how sensitive is the accuracy of the matching process to the choice of those parameters. As expected, a low parameter value may yield incorrect mappings, whereas a too high one may reject correct mappings.

We have compared the metamodel changes computed by EMF Compare to our results. We chose EMF Compare because this is a prototype completely available to compare metamodels. Table 4 shows the results of applying EMF Compare and our approach to the Petri Net (couple 0-2) and Java (couple 1.12 - 1.15) metamodels. The changes identified by both approaches are the same in the Java and Petri Net example. However, we observe that EMF Compare only identifies simple changes but not the complex changes. For example, EMF Compare identifies the references `src` and `dst` as additions (see Fig. 2). However, these actually correspond to one complex change (*Extract class*). Our solution properly recognizes such complex changes. That is why our approach is more appropriate for the purpose of model adaptation than EMF Compare.

Higher Order Transformation. We evaluate the precision of our entire process by comparing the probable *M2* model (generated by the adaptation transformation) and the expected *M2*. The comparison is done on couples whose *M1* and expected *M2* are available. In the Petri Net scenario (couple 0-2), we manually create *M1* and the expected *M2*. In the Java scenario (couple 1.12 1.13), the Java Parsers generate *M1* and the expected *M2* from a small Java program. Table 5 displays the number of lines contained in the models (*M1*, expected *M2*, and probable *M2*), and the execution time of the adaptation transformations. This table distinguishes the number of generated lines and the number of added lines. The added lines correspond to equivalences introduced by the user using AMW. The added line in Table 5 refers to the not discovered change on `parameters` reference. We compare *M2* to the expected *M2* using the Eclipse Textual Compare Editor. The models are quite similar. The elements not impacted by changes are preserved. In the Petri Net scenario, the elements conforming to `PTArc` and `TPArc` are created and correctly referenced. In the Java scenario, the elements impacted by the renaming change `(definition, type)` are properly migrated. In the Java scenario, the generated adaptation transformation is also applied to a large model (obtained by parsing the source code of GNU Classpath [31]). The number of lines of *M1* and *M2* is 69791 and the adaptation transformation consumes 38.782 s. The expected *M2* is not

Table 3: Simple and complex changes

Scenario	PetriNet		Java	
Couple of version	0-1	0-2	1.12-1.13	1.12-1.15
Num. simple changes	2	8	2	6
Num. complex changes	0	4	0	0
Des. simple changes	Restrict prop.: transition, place	Intr. class: PTArc, TPArc. Intr. prop.: in, out + Changes couple 0-1	Rename and generalize prop.: definition	Intr. class : UnresolvedAnnotation. Intr. superclass: ConstructorInvocation gets Invocation. Eliminate prop.: parameters. Generalize prop.: boundName + Changes couple 1.12-1.13
Des. complex changes		Extract class: src and dst from classes Place and Transition		

Table 4: Changes discovered by EMF Compare and our approach

Scenario	PetriNet		Java	
Couple of version	0-2		1.12-1.15	
Appr.	EMF Compare	Our approach	EMF Compare	Our approach
Simple changes	12	8	6	6
Complex changes	0	4	0	0
Total	12	12	6	6

Table 5: Model lines and adaptation transformations time

Scenario	PetriNet	Java
Num. lines M1	9	31
Num. lines expected M2	15	31
Num. lines probable M2	15	31
Num. generated lines transformation	44	74
Num. added lines transformation	0	1
Transformation Execution Time (s)	0.3	0.36

available because of bugs in the Java parser [8]. That is why the comparison is not done. However, we observe that the impacted and not impacted model elements are in the probable *M2*.

7.6 Discussion

Our approach generates model transformations to adapt models to metamodels including *breaking and resolvable changes*. The performance and precision of the matching strategies are good but may be improved by tunning the strategies. Finally, some strategies may require user assistance (e.g., user can provide/refine initial mappings or parameters).

7.7 Lessons learned

The following lessons are related to the improvement of the matching strategies in terms of precision and performance.

7.7.1 Creation heuristic selection: an agreement between performance and precision

According to our experience, when the *Creation* heuristics create too many mappings, the performance of the subsequent heuristics decreases. We could select the creation heuristics by taking into account the size of metamodels. We may apply the simple creation heuristics (e.g., *Creation by type*) on small metamodels, and the more elaborated (e.g., *Creation by type and fullname*) ones on large metamodels. We should keep in mind that even though the elaborated creation heuristics reduce the number of equivalences, those heuristics may compromise the precision of the matching process. This is the reason for the error that we get in the experimentation (see Section 7.5.2).

7.7.2 Similarity heuristics complementing each other

The precision of the matching process is low when we select only a similarity heuristic (e.g., name similarity). We mostly have to select several similarity heuristics rather than one. We can obtain fairly good approximations of equivalences and differences by executing *Similarity* heuristics than complement each

[8] We observe that not all versions of the parser actually work. This is to be expected considering we are getting them directly from a version control system, and not from stable builds.

other, i.e., heuristics that compare different properties of the KM3 concepts, for example, the *Name Similarity* and *Context similarity* heuristics.

7.7.3 Matching susceptible to parameters selection

We have experimented different thresholds and weights to understand how sensitive is the precision of the matching process to the choice of those parameters. As expected, a low parameter value may yield incorrect mappings, whereas a too high one may reject correct mappings. In addition, the selection of accurate weights may save executions of heuristics. For example, the tuning of the *Similarity by Internal Properties* weights saves executions of the most expensive heuristic: *Context Similarity* (this consumes around 85% of the total execution time when is executed more than once). In future research we hope to provide more guidelines on the parameter selection issue.

7.7.4 User-driven refinement

In our experimentation, the matching process computes a very low number of false positives. In general, the user is the one who can fully determinate whether false positives exist or not. Moreover, s/he may want to manually adjust them until getting the intended result. We provide the AMW tool to refine the computed mappings.

7.7.5 Metamodel information guides Rewriting heuristics selection

When the matching strategy only includes the *Nesting* heuristic, the gap between the generated *M2* model and the expected *M2* model may be significantly wide. We deal with this issue by executing other *Rewriting* heuristics. The selection of them may be guided by additional information about the metamodels. For example, the metamodels including complex changes (e.g., the Petri Net example) requires the *Complex Changes* heuristic. The metamodels with deep class hierarchies (e.g., the Java example) needs the *Flattening* heuristic. We need to experiment with other kinds of metamodels in order to provide more outlines.

8 Conclusion

In this report, we presented an MDE approach for adapting models to its evolving metamodel. After considering an initial classification of metamodel changes, we described how our approach generates adaptation transformations. The matching strategies computes equivalences and changes between the metamodel by executing a set of heuristics. Equivalences and differences are saved in a matching model. A HOT translates this matching model into an executable adaptation transformation. The adaptation transformations preserve unchanged model elements and migrate changed model elements. An experimentation on small (Petri Net) and large metamodels (Netbeans Java) illustrates the performance and precision of our approach. We concluded with regarding the experimentation results that new heuristics and strategies can be required to get more accurate mappings in optimal time. We are working on a Domain Specific Language (DSL) for expressing heuristics/strategies of

model matching. This DSL is supposed to make simpler the implementation of heuristics and strategies. Other interesting future work is to investigate the appropriate selection of thresholds and weights.

9 Acknowledgments

This work has been supported by two ANR projects: FLFS and OpenEmbeDD.

References

[1] Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., Jazayeri, M.: Challenges in software evolution. In: IWPSE, IEEE Computer Society (2005) 13–22

[2] Jouault, F., Bézivin, J.: KM3: a DSL for Metamodel Specification. In: Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, LNCS 4037, Bologna, Italy (2006) 171–185

[3] Hunt, J.W., McIlroy, M.D.: An algorithm for differential file comparison. Technical Report CSTR 41, Bell Laboratories, Murray Hill, NJ (1976)

[4] The MathWorks: Simulink - Simulation and model based design, `http://www.mathworks.com/products/simulink/`. (2008)

[5] Universität Siegen: The SiDiff Project, `http://pi.informatik.uni-siegen.de/Projekte/sidiff/`. (2008)

[6] Kurtev, I., Bézivin, J., Jouault, F., Valduriez, P.: Model-based DSL frameworks. In: Companion to the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, OR, USA, ACM (2006) 602–616

[7] Rahm, E., Bernstein, P.: A survey of approaches to automatic schema matching. The VLDB Journal **10**(4) (2001) 334–350

[8] Do, H.H.: Schema Matching and Mapping-based Data Integration. PhD thesis, University of Leipzig (2005)

[9] Ohst, D., Welle, M., Kelter, U.: Differences between versions of UML diagrams. SIGSOFT Softw. Eng. Notes **28**(5) (2003) 227–236

[10] Xing, Z., Stroulia, E.: UMLDiff: an algorithm for object-oriented design differencing. In: ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, New York, NY, USA, ACM (2005) 54–65

[11] Girschick, M.: Difference detection and visualization in UML class diagrams. Technical report, TU Darmstadt (2006)

[12] Treude, C., Berlik, S., Wenzel, S., Kelter, U.: Difference computation of large models. In Crnkovic, I., Bertolino, A., eds.: ESEC/SIGSOFT FSE, ACM (2007) 295–304

[13] Sriplakich, P., Blanc, X., Gervais, M.P.: Supporting collaborative development in an open MDA environment. In: ICSM, IEEE Computer Society (2006) 244–253

[14] Wenzel, S., Kelter, U.: Analyzing model evolution. In Robby, ed.: ICSE, ACM (2008) 831–834

[15] Eclipse.org: EMF Compare, `http://wiki.eclipse.org/index.php/EMF_Compare`. (2008)

[16] Wachsmuth, G.: Metamodel adaptation and model co-adaptation. In Ernst, E., ed.: Object-Oriented Programming, 21st European Conference, ECOOP 2007, Berlin, Germany, Proceedings. Volume 4609 of Lecture Notes in Computer Science., Springer (2007) 600–624

[17] Gruschko, B., Kolovos, D., Paige., R.: Towards synchronizing models with evolving metamodels. In: Workshop on Model-Driven Software Evolution, MODSE 2007, 11th European Conference on Software Maintenance and Reengineering, Amsterdam, the Netherlands. (2007)

[18] Cicchetti, A., Ruscio, D.D., Eramo, R., Pierantonio, A.: Automating co-evolution in model-driven engineering. In: EDOC 'O8: Proceedings of the 12th IEEE International EDOC Conference, München, Germany (2008)

[19] Netbeans: Java metamodels, `http://java.netbeans.org/source/browse/java/javamodel/src/org/netbeans/jmi/javamodel/resources/Attic/javamodel.xml`. (2008)

[20] Eclipse.org: MoDisco project, `http://www.eclipse.org/gmt/modisco/`. (2008)

[21] Eclipse.org: TCS project, `http://www.eclipse.org/gmt/tcs/`. (2008)

[22] Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In Jarzabek, S., Schmidt, D.C., Veldhuizen, T.L., eds.: Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA, Proceedings, ACM (2006) 249–254

[23] Jouault, F., Kurtev, I.: Transforming models with ATL. In: Proceedings of the Model Transformations in Practice Workshop, MoDELS 2005, Montego Bay, Jamaica (2005)

[24] Didonet del Fabro, M.: Metadata management using model weaving and model transformation. PhD thesis, Université de Nantes (2007)

[25] Euzenat, J., Shvaiko, P.: Ontology Matching. Springer, Heidelberg (DE) (2007)

[26] Cohen, W., Ravikumar, P., Fienberg, S.: A comparison of string distance metrics for name-matching tasks. In Kambhampati, S., Knoblock, C.A., eds.: Proceedings of Workshop on Information Integration on the Web, IIWeb 2003, Acapulco, Mexico. (2003) 73–78

[27] University of Princeton: Wordnet: An Electronic Lexical Database, `http://wordnet.princeton.edu/`

[28] Melnik, S., Garcia-Molina, H., Rahm, E.: Similarity flooding: A versatile graph matching algorithm and ist application to schema matching. In: Proc. 18th ICDE, San Jose, CA (2002)

[29] Didonet Del Fabro, M., Bézivin, J., Jouault, F., Breton, E., Gueltas, G.: AMW: A generic model weaver. In: Proceedings of the 1ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM05). (2005)

[30] Melnik, S.: Generic Model Management: Concepts and Algorithms. PhD thesis, University of Leipzig (2004)

[31] GNU: Classpath source code. (`http://ftp.gnu.org/gnu/classpath/`)

Contents

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803