# Managing Model Adaptation by Precise Detection of Metamodel Changes

Kelly Garcés, Frédéric Jouault, Pierre Cointe, Jean Bézivin

## HAL Id: hal-00466940

## https://hal.archives-ouvertes.fr/hal-00466940

Submitted on 25 Mar 2010

# Managing Model Adaptation by Precise Detection of Metamodel Changes

Kelly Garcés[1,2], Frédéric Jouault[1], Pierre Cointe[2], and Jean Bézivin[1]

[1] AtlanMod, EMN-INRIA Rennes
[2] ASCOLA, LINA (UMR 6241)-INRIA Rennes
{kelly.garces,pierre.cointe}@emn.fr
{frederic.jouault,jean.bezivin}@inria.fr

**Abstract.** Technological and business changes influence the evolution of software systems. When this happens, the software artifacts may need to be adapted to the changes. This need is rapidly increasing in systems built using the Model-Driven Engineering (MDE) paradigm. An MDE system basically consists of metamodels, terminal models, and transformations. The evolution of a metamodel may render its related terminal models and transformations invalid. This paper proposes a three-step solution that automatically adapts terminal models to their evolving metamodels. The first step computes the equivalences and (simple and complex) changes between a given metamodel, and a former version of the same metamodel. The second step translates the equivalences and differences into an adaptation transformation. This transformation can then be executed in a third step to adapt to the new version any terminal model conforming to the former version. We validate our ideas by implementing a prototype based on the AtlanMod Model Management Architecture (AMMA) platform. We present the accuracy and performance that the prototype delivers on two concrete examples: a Petri Net metamodel from the research literature, and the Netbeans Java metamodel.

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

## 1 Introduction

Software engineers usually have to adapt computer systems to technological and business changes. This need is rapidly increasing in systems built using the Model-Driven Engineering (MDE) paradigm. An MDE system basically consists of metamodels, terminal models, and transformations. The addition of new features and/or the resolution of bugs may change metamodels. The changes may break the consistency of related terminal models and transformations. In this work, we focus on terminal models consistency. Fig. 1 illustrates the problem: a metamodel *MM1* evolves into a metamodel *MM2* (see the dotted arrow). Our concern is to adapt any terminal model *M1* conforming to *MM1* to the new metamodel version *MM2* (see the dashed arrow).

This paper proposes a three-step adaptation. 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 discovered equivalences and changes. Finally, this transformation brings *M1* into agreement with *MM2*, and persists the result in *M2*.

The bulk of this work is devoted to the first step that discovers equivalences, as well as simple and complex changes. We explicitly distinguish two kinds of changes because complex changes need a more insightful adaptation that simple changes. Whereas a simple change describes the addition, deletion, or update of one metamodel concept, a complex change integrates a set of actions affecting multiple concepts [3]. The paper reports a family of heuristics responsible for figuring out both simple and complex changes, and representing them in a straightforward way.
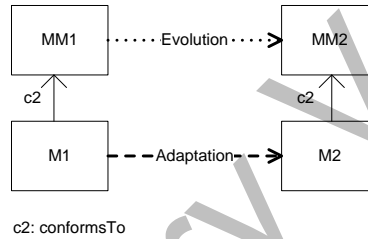


**Fig. 1.** Metamodel evolution and model adaptation

We have implemented a proof-of-concept prototype based on the AtlanMod Model Management Architecture (AMMA) platform [2]. We have evaluated its performance and accuracy on two examples: a Petri Net metamodel from the research literature, and the Netbeans Java metamodel. The Petri Net metamodel is selected because it is simple enough to be analyzed in a paper, and includes complex changes. The concrete choice of the Netbeans Java metamodel is driven by three main reasons: 1) it is a "real-life" problem, 2) experimental data is widely available (open-source), and 3) the metamodel and terminal models are significantly larger than those of Petri Net, which illustrates the scalability of our approach.

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 prototype, we are able to analyze on a desktop machine any pair of the Petri Net metamodels in under 1 second, and any pair of the Java metamodels in under 10 seconds. Moreover, our tool always discovers the changes, and only fails by identifying simple changes when in truth there is an equivalence (in 1% of the cases).

---

[3] The reader interested on examples of simple and complex changes may consult [1].

This paper is organized as follows: Section 2 compares our contributions to other known solutions. Section 3 presents a running example. Section 4 presents our solution to adapt models to evolving metamodels. Section 5 describes the results of applying our approach on the examples. Finally, Section 6 concludes the paper.

## 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.

We now describe the related works closer to the first problem. In the context of relational and object-oriented data bases, the production of equivalences between two schemas/ontologies has been invested in [3][4]. In the MDE domain, the approaches of [5][6][7][8] present algorithms for detecting changes between UML models. Sriplakich et al. [9] identify simple changes in terminal models conforming to any metamodel. Wenzel et al. [10] 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. The EMF Compare tool [11] reports simple changes between terminal model pairs or metamodel pairs. Finally, Falleri et al. [12] automatically detect equivalences between two metamodels using the algorithm *Similarity Flooding* described in [13].

In contrast to the first issue, the second one has been addressed by some recent approaches. The works described in [14][15][16][17][18] assume traces of changes are available, and derive adaptation transformations from them. In particular, [14], [18], and [17] apply stepwise automatic transactions on *MM1* to obtain *MM2*. These approaches then reuse the logs of applied transactions to derive adaptation transformations. Cicchetti et al. [16] use difference models provided by external tools.

The following six items position our approach in comparison with the solutions mentioned above:

1. Similarly to [10][18], our approach computes equivalences and differences between any pair of metamodels (e.g., representing schemas, UML models, ontologies, grammars) .
2. Our solution overlaps the solutions presented in [14][16][17] in the sense of considering both simple and complex changes.
3. As in [12], in our matching process the robust algorithm *Similarity Flooding* can be executed. Falleri's main contribution is to provide 6 graph representation configurations. These configurations generate graphs that contain some metamodel information or all the metamodel information. Although light metamodel representations benefits to the algorithm performance, they point out the matching process accuracy increases when all the metamodel information is represented in the graphs. This is what our approach exactly does.

4. Unlike existing approaches [14][15][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.

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

6. An experimentation shows that our approach scales to larger metamodels and models. This is an improvement on other techniques developed to date.

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 metamodel evolution. In contrast, we propose a solution that addresses all the described model adaptation issues in a consistent and integrated way.

## 3   Running example

This section describes a running example, i.e., the Petri Net metamodel, and how to represent it using the KM3 metametamodel. We omit describing the Netbeans Java metamodel to save space, this is fully depicted in our technical report [1]. We choose the KM3 notation because it is simple but expressive enough to represent metamodels [19].
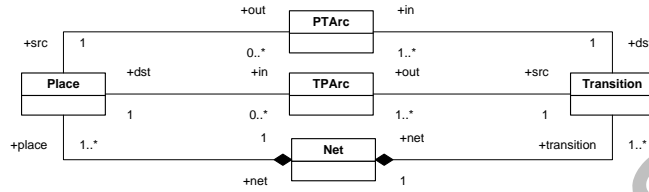
### 3.1   A sample Petri Net metamodel

This example is based on the six versions of the Petri Net metamodel provided by [14]. Fig. 2 illustrates versions 0 (*MM1*) and 2 (*MM2*). *MM1* represents simple Petri Nets. These nets may consist of any number of places and transitions. A 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:

– References `place` and `transition` change their multiplicity from 0-* to 1-*.
– Classes `PTArc` and `TPArc` as well as references `in` and `out` are added.
– References `src` and `dst` are extracted from classes `Place` and `Transition`.

*Remark 1.* The extraction of the reference `dst` illustrates a complex change named *Extract class*. This implies to add and remove a reference, add a class, and associate classes. In considering these actions as isolated simple changes, we may skip changes without migrating involved data from *M1* to *M2*. In contrast, when we distinguish the complex change, we infer (for instance) that the added property (e.g., `dst`), contained in the new class `PTArc`, actually corresponds to the property `dst` removed from the class `Place`. Since we know the relationship between the properties we can migrate the data. We thus need to explicitly distinguish complex changes in order to properly derive adaptation transformations.
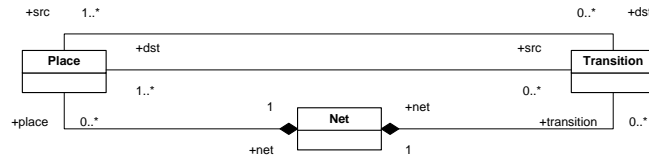
(a) Petri Net *MM1* (version 0)



(b) Petri Net *MM2* (version 2)

**Fig. 2.** Petri Net metamodels

## 3.2 The KM3 metametamodel

Fig. 3 shows the basic concepts of the KM3 metametamodel. 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 Petri Net metamodel, `Place` conforms to the `Class` concept. The reference `dst` conforms to the `Reference` concept, this has the attributes `lower` and `upper` with value `0` and `*`.
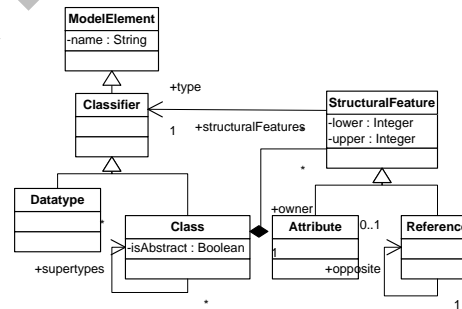


**Fig. 3.** KM3 concepts

## 4    A model adaptation approach

As we introduced earlier, our model adaptation approach adapts terminal models in three steps (Fig. 4). In the first step, a *matching strategy* computes equivalences and differences between the metamodels *MM1* and *MM2* by executing a set of heuristics (available in a library). Equivalences and differences are represented by a *matching model*. In the second step, the matching model is translated into an adaptation transformation by using a Higher-Order Transformation (HOT). Finally, the adaptation transformation is executed. Below we discuss the three steps in detail.
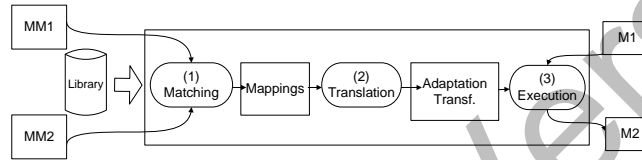


**Fig. 4.** Approach Overview

### 4.1    Matching equivalences and differences

**Matching model** Before describing the matching step, we explain how matching models represent equivalences and differences. A matching model conforms to the *Matching metamodel*[4] illustrated in Listing 1.1. The main concept is Equal which describes a mapping (or correspondence) between an element of *MM1* (leftElement) and an element of *MM2* (rightElement). Equal has a similarity value (between 0 and 1) that represents the plausibility of the correspondence. An equivalence with similarity value 1 represents that the *MM2* element is an identical copy of the *MM1* element. An equivalence with similarity value 0.7 describes that the *MM2* element is a copy of the *MM1* element including simple modifications. An equivalence with similarity value 0 link elements different to each other. Other basic concepts are Added and Deleted which mark a metamodel element as deleted/added from/into *MM1*.

Equal, Added, and Deleted can be extended to describe more specific equivalences or changes. For example, EqualClass, EqualStructuralFeature, Equal-Reference, EqualAttribute indicate the (KM3) types of leftElement and rightElement. The concept AssociatedClassExtracted, in order, links properties undergoing the change *Extract class*.

**Listing 1.1.** Excerpt of the matching metamodel

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

---

[4] This metamodel extends the core weaving metamodel proposed by [20].

```
3    class Link extends WLink {
4      reference left[0−1] container : LeftElement;
5      reference right[0−1] container : RightElement;
6    }
7    class Equal extends Link {
8      attribute similarity : Double;
9    }
10   class Added extends Link {}
11   class Deleted extends Link {} ...
12   class AssociatedClassExtracted extends EqualStructuralFeature {
13     reference associatedReference container : RightElement;
14   }
```

**Matching step** Matching models are computed by matching strategies, i.e., processes that incrementally execute a set of heuristics. The heuristics are needed because the comparison of metamodels (graphs) is a NP complete problem [21]. Fig. 5 is a simple overview of what a matching strategy is. There may be more elaborated matching strategies, e.g., including loops. Fig. 5 presents the kinds of heuristics (using a particular symbol) and their execution order. Every heuristic produces a *matching model*. For the sake of simplicity, we omit intermediate matching models in Fig. 5.

Fig. 5 shows a *Creation* heuristic which prepares a collection of equivalences by matching the elements of *MM1* and *MM2*. Afterward, *Similarity* heuristics compute similarity values by comparing the names, internal properties, and structures of the matched elements. Subsequently, *Filtering* heuristics select equivalences taking into account the confidence value computed by the *Similarity* heuristics. A *Differentiation* heuristic recognizes equivalences, additions, and deletions. The matching step finishes when *Rewriting* heuristics reorganize a given matching model to make it closer to adaptation transformations. Note that the user can build (tune) matching strategies by choosing concrete heuristics (for each kind) from the available library. Moreover, s/he can refine the matching models generated along the process.

Let us now conceptualize the kinds of heuristic presented in the previous paragraph. Each category contains a set of particular heuristics. We select these heuristics because they analyze "safe" indicators (e.g., name) that two elements are the same [6]. In this paper, we describe the heuristics in terms of the KM3 concepts. Our approach remains nonetheless generic (i.e., independent of KM3). The heuristics can be implemented using other formalisms such as MOF or EMF/Ecore. We now illustrate the family of heuristics using the AtlanMod Transformation Language (ATL)[22]. An ATL transformation takes models (conforming to input metamodels) as input, and yields models (conforming to output metamodels) as output. The transformations are composed of rules. A rule consists of two mandatory parts: the *from*, and the *to* [23]. The *from* part defines an input pattern and an Object Constraint Language (OCL) condition [24] (written in terms of input metamodel concepts). The *to* part specifies an output pattern and bindings. Each category below has a code listing whose number is enclosed between parenthesis.
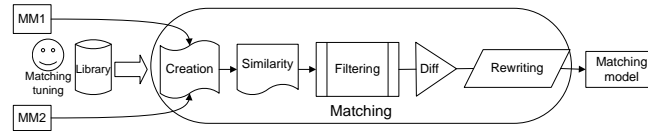
**Fig. 5.** A simple matching strategy

*Creation* (Listing 1.2) The creation heuristics match concepts of *MM1* (line 3) and concepts of *MM2* (line 4), and create equivalences (lines 7-10) when the concepts hold a condition (line 5). The properties `left` and `right` of `Equal` refer to *MM1* and *MM2* (lines 8-9). The condition is written in terms of the KM3 metamodel concepts.

**Listing 1.2.** Creation transformation excerpt

```
1  rule Creation {
2    from
3      a : KM3!<Concept> in MM1
4      b : KM3!<Concept> in MM2
5      (condition) -- for example , a.type = b.type
6    to
7      e : EqualMM!Equal (
8        left <- a.ref ,
9        right <- b.ref
10     )
11 }
```

A simple *Creation* heuristic is *Creation by type*. This creates a mapping when two elements conform to the same KM3 type (i.e., `Class`, `Reference` or `Attribute`) (see comment in line 5). Another *Creation* heuristic is *Creation by Type and FullName* which creates mappings when two elements has the same KM3 type and fullname. The fullname is a string that concatenates the names of elements related to another element. For example, the fullName of `transition` reference is `PetriNet|Net|transition` because `PetriNet` package contains `Net` class, and this class contains `transition` reference.

*Similarity* (Listing 1.3) The similarity heuristics compute a similarity value for each equivalence prepared by *Creation* heuristics (line 3). A function (*func*) establishes the similarity values (line 6). Note that the *Similarity* heuristics have no longer the KM3 concepts in the input pattern. Instead of that, the Equal concept is used. The function refers to *MM1* and *MM2* concepts by using `left` and `right` of `Equal`.

**Listing 1.3.** Similarity transformation excerpt

```
1  rule Similarity {
2    from
3      e : EqualMM!Equal
4    to
5      e : EqualMM!Equal (
6        sim <- func
7      )
8  }
```

Next we present four *Similarity* heuristics whose functions calculate similarity values by comparing particular properties of the KM3 metamodels.

- *Name Similarity* compares the names of KM3 elements in different ways, for instance, using string comparison algorithms [25] or dictionaries of synonyms [26].
- *Multiplicity Similarity* compares the multiplicity of references and attributes. This assigns a similarity value to mappings connecting metamodel references that have the same multiplicity (lower and upper bounds).
- *Similarity by Internal Properties* compares several properties of the KM3 elements. Each property contributes a relative similarity value, i.e., the similarity value multiplied by a weight. The net similarity value is the sum of all relative values. For instance, the net similarity value of elements conforming to `Reference` is the sum of the relative values of `names`, `types`, `multiplicities`, and `opposites`.
- *Context Similarity* compares the relationships between metamodel elements. For example, this compares attributes/references contained in a given class, its superclasses, and its associated classes. The implementation of this *Similarity* heuristic is more complex than the previously presented ones. This is inspired from the *Similarity Flooding* (SF) algorithm. Our algorithm is executed in two steps. The first step associates two equivalences (*e1* and *e2*) if there is a relationship between the linked elements. The second step propagates the similarity value from *e1* to *e2* because of the relationship.

*Filtering* (Listing 1.4) The previous heuristics may have created unwanted equivalences (i.e., equivalences with low similarities). The filtering heuristics select the equivalences (line 3) whose similarity values satisfy a condition (line 4). A basic filtering heuristic is *Threshold*. This selects the mappings with a similarity value higher than a given threshold value (see comment of line 4).

**Listing 1.4.** Filtering transformation excerpt

```
1 rule Filtering {
2   from
3     e : EqualMM!Equal
4       (condition) --For example, e.similarity > threshold
5   to
6     e : EqualMM!Equal
7 }
```

*Differentiation* (Listing 1.5) This kind of heuristic distinguishes between equivalent, deleted, and added metamodel elements. A concrete implementation of *Differentiation* heuristics compares KM3 elements to equivalences. The intuition is that not linked metamodel elements correspond to deletions and additions. Listing. 1.5 marks *MM1* elements as deleted elements (line 6) when they are not linked by equivalences (lines 3-4). The implementation contains another rule, ommitted because of space constraints, that marks unlinked *MM2* elements as added elements.

**Listing 1.5.** Differentiation transformation excerpt

```
1 rule Deleted {
2   from
3     a : KM3!ModelElement in MM1
4       a.notLinked()
5     )
6   to
7     e : EqualMM!Deleted
8 }
```

*Rewriting* (Listing 1.6) Before this step, most equivalences and differences are contained in a single large collection. This heuristic reorganizes/retypes the equivalences and differences in order to make them more semantically richer. We discern three *Rewriting* heuristics: *Nesting*, *Flattening*, and *Complex changes*. The *Nesting* and *Flattening* heuristics reorganize the equivalences considering the relationships (containment and inheritance) between the linked concepts. For instance, the *Nesting* heuristic rewrites (`transition`, `transition`) as a child of (`Net`, `Net`) because of the containment relationship between these elements. The *Complex change* heuristic infers complex changes from equivalences, additions, and deletions. For example, Listing 1.6 shows a rule that verifies if change *Extract Class* has happened. The rule assemblies two properties $a$ (added) and $d$ (deleted) using the `AssociatedClassExtracted` type. The conditions are: 1) an introduced class owns the property $a$ (line 5), and 2) this class is associated to other class that contains $d$ (line 7).

**Listing 1.6.** Complex changes transformation excerpt

```
1 rule AssociatedClassExtracted {
2   from
3     d : EqualMM!DeletedStructuralFeature,
4     a : EqualMM!AddedStructuralFeature (
5           a.right.target.owner.isNewClass()
6           and
7           a.right.target.owner.isAssociatedTo(d.left.target.owner)
8       )
9   to
10     e : EqualMM!AssociatedClassExtracted
11 }
```

### 4.2   Translation to adaptation transformations

In this step, the equivalences and differences are translated into an executable adaptation transformation via a HOT. The HOT takes as input the final matching model, and generates as output a model transformation written in a particular transformation language (e.g., ATL, XSLT, SQL-like). The HOT follows the guidelines below:

- Yield a transformation rule for each `EqualClass` that links no abstract classes. The HOT takes referred left and right classes to yield input and output patterns.
- Create a binding for each `EqualStructuralFeatures` attached to a `Equal-Class`. The binding complexity depends on the `Equal` type. While a simple

`EqualStructuralFeature` generates a simple binding, `EqualStructuralFeature` extensions (e.g., `AssociatedClassExtracted`) generate more elaborated bindings. In general, sophisticated bindings instruments the code that adapt *M1* models to complex changes.

Listing. 1.7 shows an adaptation transformation, written in ATL, which is generated by a concrete HOT. This creates the transformation rule `Place2Place` (line 1) from the equivalence (`Place`, `Place`). The *from* part matches the elements conforming to `Place` (line 3). The *to* part creates elements conforming to `Place`. The HOT moreover generates a complex binding (see line 6) from the equivalence (`out`, `dst`). The binding calls an additional rule (i.e., `dstPTArc`) to initialize `dst` of `PTArc` (lines 18) using the values `dst` of `Place`.

**Listing 1.7.** Transformation excerpt (Petri Net example)

```
1  rule Place2Place {
2    from
3      pV1 : MM1!Place
4    to
5      pV2 : MM2!Place (
6        out <- pV1.dst -> collect (tV1 | thisModule.dstPTArc(tV1, pV1)}))
7      )
8  }
9  unique lazy rule dstPTArc {
10   from
11     transition : MM1!Transition ,
12     place : MM1!Place
13   to
14     tV2 : MM2!PTArc (
15     dst <- transition
16   )
17 }
```

### 4.3 Adaptation transformation execution

This step simply executes the generated adaptation transformation. The transformation takes any terminal model *M1* and generates a terminal model *M2*.

## 5 Experimental validation

Section 5.1 describes the prototype platform. Section 5.2 presents the experimental settings including dataset and procedure. Section 5.3 provides the metrics to evaluate the results. Section 5.4 discusses the experimentation results. Finally, Section 5.5 shows the results of applying the EMF Compare tool to the running examples, and compares them to our results.

### 5.1 Prototype implementation

We implement the prototype on the AMMA platform [2]. More specifically, we use the AtlanMod Model Weaver (AMW) [27] to work with 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 4.1.

## 5.2   Experimental settings

**Data set**  We have results from experimentations which use 8 versions of the Netbeans Java metamodel, and 6 versions of a Petri Net metamodel provided by [14]. For the sake of readability, we just present the results in applying our approach on three versions of each metamodel. These versions are chosen because they contain significant changes. In the Java example, we choose the versions 1.12, 1.13, and 1.15. In the Petri Net example, we use the versions 0, 1, and 2. Table 1 shows the number of elements (classes, attributes and references) contained in the versions. We match the following couples of versions: $0 - 1$, $0 - 2$, $1.12 - 1.13$, and $1.12 - 1.15$.

**Table 1.** Metamodel elements

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

**Procedure**  We tested different matching configurations until obtaining the strategies more suitable for the examples. Garces et al. [1] presents lessons learned from this selection process. We have picked up the matching strategies ($A$ and $B$) for matching the Petri Net metamodels and the Java metamodels, respectively. The heuristics that include each strategy are:

– **Matching Strategy A**: Creation by type, Similarity by internal properties, Context similarity, Threshold, Differentiation, Nesting, and Complex changes.
– **Matching Strategy B**: Creation by type and fullname, Similarity by internal properties, Context similarity, Threshold, Differentiation, Nesting, and Flattening.

This selection should not question the applicability of our approach, but show that the matching accuracy and performance highly depends on the metamodels.

## 5.3   Metrics

We have measured the matching step accuracy by applying three metrics [28]: $Precision(x) = \frac{CorrectFound(x)}{TotalFound(x)}$, $Recall(x) = \frac{CorrectFound(x)}{TotalCorrect(x)}$, and $Fscore(x) = \frac{2*Recall(x)*Precision(x)}{Recall(x)+Precision(x)}$.

The $x$ denotes equivalences, additions/deletions, or complex changes. Besides additions and deletions, we have not evaluated other simple changes because these require no elaborated adaptation transformations. The expected values of these metrics are between 0 and 1. The higher is the precision value, the smaller is the set of wrong mappings. The higher is the recall value, the smaller is the

set of the mappings that have not been found. Fscore is a global measure of the matching quality. A high fscore value indicates a matching of high quality.

We have identified the correct equivalences and changes in two ways. In the Petri Net example, we manually discovered the changes. In the Java example, we relied on the changes logged in the Netbeans repository. We also considered other manually discovered changes. We remarked that some repository logs do not report all the performed changes.

Besides matching accuracy, we have measured the matching process performance. This has been executed on a machine with Intel Core 2 Duo (2.4 GHz) and 1GB RAM.

### 5.4   Results

**Matching accuracy** Fig. 6 gives the prototype's accuracy. The histograms display measures (precision, recall, fcore) for each selected couple of version. The three bars (from left to right) show the accuracy of equivalences, additions/deletions, and complex changes. Some bars are missing because certain couples of versions contain no deletions/additions or complex changes.

The results show that the prototype achieves a high accuracy not only in detecting the correct equivalences, additions/deletions, but also in detecting complex changes. Taking fscore as an example, the percentage of correct equivalences, and additions and deletions ranges from 99%-100%, and 90%-100%, respectively. Averanging accross all experiments, the fscore of complex changes is 100%. In particular, our prototype fails in identifying additions/deletions instead of equivalences (1% of cases).

Even though we have applied our approach to only a small number of test cases (mostly because of model availability restrictions), this number is significant in comparison with test cases of other closely related approaches like [14][15][16][17][18]. We are looking for other benchmarks in open-source projects, and we hope to provide more validation material in the future.

**Performance** In the Petri Net example, the matching process consumes less than 1 second. In the Java example, the matching process approximately takes 10 seconds. A table containing the execution times of the heuristics in detail can be find in [1]. Even if the matching step consumes a relevant amount of resources, we should remember that this process generates an adaptation transformation that can be used several times.

### 5.5   EMF Compare versus our approach

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 2 shows the fscore that EMF Compare and our approach, denoted by i. and ii., deliver on the Petri Net (couple 0-2) and Java (couple 1.12 - 1.15) examples. While EMF Compare is fairly good for identifying additions and deletions, this fails in rendering them as isolated actions. Because
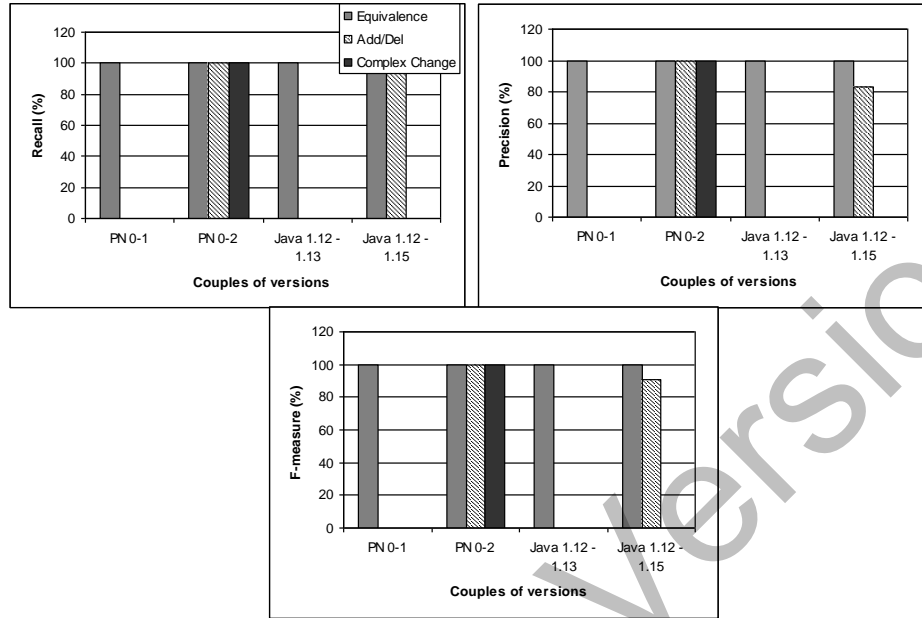
**Fig. 6.** Matching accuracy results

model adaptation automation needs to distinguish complex changes (i.e., not only simple changes), our approach is more appropriate for this purpose than EMF Compare.

**Table 2.** Fscore EMF Compare (i.) - Our approach (ii.)

| Example | PetriNet | | Java | |
|---|---|---|---|---|
| Couples of versions | 0-2 | | 1.12-1.15 | |
| Approach | i. | ii. | i. | ii. |
| Additions-Deletions | 0.8 | 1 | 1 | 0.9 |
| Complex changes | 0 | 1 | 0 | 0 |

## 6    Conclusions

In this paper, we presented an MDE approach for adapting models to their evolving metamodel. Matching strategies compute equivalences and changes between two metamodels by executing a set of heuristics. These equivalences and differences are saved in a matching model. A Higher-Order Transformation translates this matching model into an executable adaptation transformation. We reported the performance and precision of our approach which are pretty good, and may

be even further improved by means of tuning. We also compared our solution
to related works, and we showed that no other work known to us covers fully
the problem as we identified it (e.g., most other works only cover evolution
with available trace of changes). Moreover, our validation covers a wider spec-
trum than existing works: a relatively simple case from the literature (i.e., a
Petri Net metamodel), but also a real-life scenario (i.e., a Java metamodel). We
have used the family of heuristics to design the constructs of the AtlanMod
Matching Language (AML), a Domain-Specific Language (DSL) for expressing
matching strategies [29]. AML allows to express not only strategies to match
two metamodels, but also any pair of models. By using this language, we hope
to implement matching strategies more easily, and extract further guidelines on
proper parametrization of them.

## References

1. Garcés, K., Jouault, F., Cointe, P., Bézivin, J.: Adaptation of models to evolving
   metamodels. Technical report, INRIA (2008)
2. 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
3. Rahm, E., Bernstein, P.: A survey of approaches to automatic schema matching.
   The VLDB Journal **10**(4) (2001) 334–350
4. Do, H.H.: Schema Matching and Mapping-based Data Integration. PhD thesis,
   University of Leipzig (2005)
5. Ohst, D., Welle, M., Kelter, U.: Differences between versions of UML diagrams.
   SIGSOFT Softw. Eng. Notes **28**(5) (2003) 227–236
6. Xing, Z., Stroulia, E.: UMLDiff: an algorithm for object-oriented design differenc-
   ing. In: ASE '05: Proceedings of the 20th IEEE/ACM international Conference on
   Automated software engineering, New York, NY, USA, ACM (2005) 54–65
7. Girschick, M.: Difference detection and visualization in UML class diagrams. Tech-
   nical report, TU Darmstadt (2006)
8. 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
9. Sriplakich, P., Blanc, X., Gervais, M.P.: Supporting collaborative development in
   an open MDA environment. In: ICSM, IEEE Computer Society (2006) 244–253
10. Wenzel, S., Kelter, U.: Analyzing model evolution. In Robby, ed.: ICSE, ACM
    (2008) 831–834
11. Eclipse.org: EMF Compare, `http://wiki.eclipse.org/index.php/EMF_Compare`.
    (2008)
12. Falleri, J.R., Huchard, M., Lafourcade, M., Nebut, C.: Metamodel matching for au-
    tomatic model transformation generation. In Czarnecki, K., Ober, I., Bruel, J.M.,
    Uhl, A., Völter, M., eds.: MoDELS. Volume 5301 of Lecture Notes in Computer
    Science., Springer (2008) 326–340
13. 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)

14. 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

15. 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)

16. Cicchetti, A., Ruscio, D.D., Eramo, R., Pierantonio, A.: Automating co-evolution in model-driven engineering. In: EDOC '08: Proceedings of the 12th IEEE International EDOC Conference, München, Germany (2008)

17. Herrmannsdoerfer, M., Benz, S., Juergens, E.: Automatability of coupled evolution of metamodels and models in practice. In Czarnecki, K., Ober, I., Bruel, J.M., Uhl, A., Völter, M., eds.: MoDELS. Volume 5301 of Lecture Notes in Computer Science., Springer (2008) 645–659

18. Vermolen, S.D., Visser, E.: Heterogeneous coupled evolution of software languages. Lecture Notes in Computer Science **5301** (September 2008) 630–644 In K. Czarnecki and I. Ober and J.-M. Bruel and A. Uhl and M. Voelter (eds.) Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MODELS 2008). Toulouse, France, October 2008.

19. 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

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

21. Przulj, N., Corneil, D.G., Jurisica, I.: Efficient estimation of graphlet frequency distributions in protein-protein interaction networks. Bioinformatics **22**(8) (2006) 974–980

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

23. Eclipse.org: The ATL User Manual, `http://www.eclipse.org/m2m/atl/doc/ATL_User_Manual[v0.7].pdf`. (2008)

24. OMG: OCL 2.0 Specification, OMG Document formal/2006-05-01, `http://www.omg.org/docs/ptc/05-06-06.pdf`. (2006)

25. 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

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

27. 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)

28. Rijsbergen, C.J.V.: Information Retrieval. Butterworths (1979)

29. Garcés, K., Jouault, F., Cointe, P., Bézivin, J.: A Domain Specific Language for Expressing Model Matching. In: Proceedings of the 5ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM09). (2009)