MatchBox: Combined Meta-model Matching for Semi-automatic Mapping Generation

Konrad Voigt[*]
SAP Research CEC Dresden
Chemnitzer Str. 48
01187 Dresden, Germany
konrad.voigt@sap.com

Petko Ivanov
SAP Research CEC Dresden
Chemnitzer Str. 48
01187 Dresden, Germany
p.ivanov@sap.com

Andreas Rummler
SAP Research CEC Dresden
Chemnitzer Str. 48
01187 Dresden, Germany
andreas.rummler@sap.com

ABSTRACT

Data integration is a well-known challenge offering a common view on heterogeneous data, e. g. to ensure consistency or tool interoperability. To implement this integration MDE proposes to apply model transformations. A model transformation requires meta-model matching, i. e. the task of discovering semantic correspondences between elements. Recently, semi-automatic matching has been proposed to support transformation development by mapping generation.

However, current meta-model matching approaches concentrate on a fixed non-configurable set of matching algorithms and often miss a thorough evaluation, thus no estimate concerning their quality can be made. We tackle these issues by proposing MatchBox, an approach using a configurable combination of matchers in a common framework. Thereby, we adapt and extend established schema matching techniques for the purpose of meta-model matching.

Additionally, we present a benchmark for meta-model matching consisting of 23 real-world model transformations and mappings. This benchmark is used to comprehensively evaluate MatchBox. The results obtained lead to conclusions regarding our approach and the effectiveness of meta-model matching.

Categories and Subject Descriptors

D.2.11 [**Software architectures**]: Domain-specific architectures; D.2.12 [**Interoperability**]: Data mapping

General Terms

Algorithms, Standardization, Languages

[*]This work was done in context of a doctoral work at the chair of Prof. Uwe Aßmann at the TU Dresden. The work was funded by means of the German Federal Ministry of Economy and Technology under the promotional reference "01MQ07012". The author takes the responsibility for the contents.

Keywords

Model engineering, meta-model matching, mapping generation

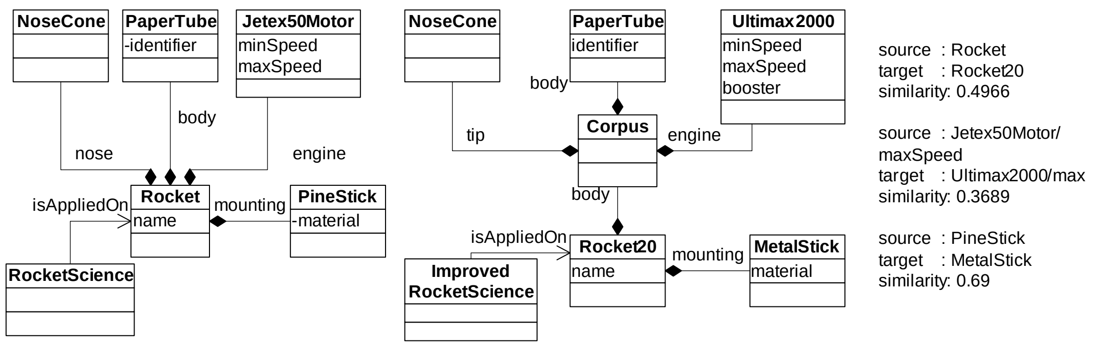
1. INTRODUCTION

The well-known problem of data integration involves challenges like synchronisation and tool interoperability. A prerequisite is the task of data set matching that describes the search for semantic correspondences between elements, i. e. mappings. In order to cope with this problem semi-automatic schema matching [15] has been proposed which developed during the last years to an established field in XML- and database schema integration. Complementary, matching has been adopted in ontology matching [2] and in meta-model matching. Since Model Driven Engineering proposes to use model transformation for the integration [3], matching as in meta-model matching [12, 9, 6, 7] aims to support the development of model transformations. These transformations are the implementation of mappings and hence perform the actual integration. Consequently, a usual meta-model matching system receives two meta-models as input and creates mappings or most preferably transformations as output.

These approaches use different information, such as labels, structure, etc., and combine them in a fixed manner for deriving a mapping. Unfortunately, all these approaches miss an thorough evaluation. Their quality is evaluated by applying up to five (synthetic) examples, tailored towards meta-model matching.

In this paper we demonstrate an application of a combination approach on meta-model matching. Thereby, we propose MatchBox, a framework combining different matching techniques. We present an adoption of schema matching techniques to meta-model matching by transforming a meta-model's graph structure into a tree. Applying these techniques we present an evaluation of MatchBox based on 23 real-world transformations and mappings. Building upon our results we identify drawbacks and possible extensions for MatchBox.

We structure our paper as follows; in Sect. 2 we discuss related work and the subsequent section 3 gives an example to illustrate the task of meta-model matching. This is followed by section 4 presenting the architecture of MatchBox, its matching process and its configuration. It further identifies issues of adopting a schema matching approach for meta-models. Our evaluation is given in Sect. 5 presenting our results regarding MatchBox's quality. We conclude our paper by giving a summary and an outlook on future work.



source : Rocket
target : Rocket20
similarity: 0.4966

source : Jetex50Motor/maxSpeed
target : Ultimax2000/max
similarity: 0.3689

source : PineStick
target : MetalStick
similarity: 0.69

(a) Simple Rocket Meta-model (b) Simple Rocket 2.0 Meta-model (c) Sample Mapping Output

Figure 1: Example for two meta-models and MatchBox output and architecture of MatchBox

2. RELATED WORK

Meta-model matching originates from the field of schema matching in data bases and XML [13] which has also been adopted by ontology matching (also called ontology alignment). Both, schema and ontology matching approaches are based on a different formalism than meta-models. They do not consider meta-model specific aspects, i. e. meta-model flattening, containment semantics, inheritance, etc. or an import from a meta-model to a tree. For reviews on these matching approaches refer to [13, 15] and [2] respectively. In the following we will focus on related work in the area of meta-model matching, outlining the respective approach and its evaluation.

Meta-model matching has been done first by Lopes et al. [12] in their system called SAMT4MDE. They apply a fix-point computation based approach for determining similarity between meta-models. According to [12], their approach seems to be label-based. It uses information from names, data types and enumerations to propagate computed similarity through containment child elements. The similarity result is: different, similar, and equal for a match. There is neither a combination nor a configuration of matchers possible. Their evaluation is based on UML, Java, Ecore and C# with manually created gold-standards. Their experiments lead to results comparable to us (F-Measure 0.73) even so they used purpose-built examples.

Kappel et al. also applied meta-model matching [9]. They transformed two meta-models into corresponding ontologies and applied ontology alignment approaches. Based on five examples (UML 1.4, UML 2.0, Ecore, EER and WebML) with manually created gold-standards they used COMA++ for their evaluation. These examples were reused in our evaluation. They conclude that COMA++ performs best compared to other ontology matching approaches. The best matching results were obtained for model evolution. The values for other scenarios obtained by their evaluation are close to ours, since we reused their examples, but lower (F-Measure 0.26). They did not compute the best strategies and lost the containment semantics which is omitted by their meta-model to ontology transformation. Thereby, they did not evaluate different meta-model import strategies as we did.

Fabro and Valduriez [6] also tackled the problem of meta-model matching using a similarity flooding approach, thus allowing no matcher combination. However, their similarity algorithm can be applied sequentially, but without any weight or parameters. For their experiments they used two synthetic examples. They do no explicitly show the examples' structure and due to the purpose-built nature of their examples no reliable comparison can be made.

According to the approach proposed by Falleri et al. [7], meta-models to be matched are converted into directed labeled graphs. These graphs are used to apply a similarity flooding algorithm, whereby the similarity computation is done via label-based similarity. They apply four adjusted sample meta-models (Ecore, MiniJava, Kermeta and UML) leading to best results in mappings for MiniJava. Again, they do not consider real-world examples in a broad range and allow no combination of matchers.

To conclude, the current meta-model matching approaches rely on fix-point computation and do not provide a flexible matcher combination as we do. Additionally, meta-model matching in its current stage misses a clear foundation regarding evaluation and effectiveness in contrast to schema and ontology matching. In these fields, many approaches arose and where evaluated using a broad range of real-world data. However, all of them rely on a manual creation of gold-standards whereas we propose to have an automatic approach.

3. META-MODEL MATCHING EXAMPLE

Before explaining the architecture and the evaluation of our matching system we give an example to illustrate the task of meta-model matching. We chose an example from the field of toy rocket construction, i. e. meta-models describing a toy rockets parts. Figure 1 depicts the meta-models of our example for meta-model evolution: (a) Rocket to (b) Rocket 2.0. The task is to map between two versions of one model, which is quite common in software engineering.

A rocket consists of a NoseCone, Papertube and Jetex50-Motor and is identified by its name. The Papertube has an own identifier for retrieving it from a pile of other cones. The Jetex50Motor is described by its minSpeed and maxSpeed. A rocket is mounted onto a PineStick, which is used for starting. In order to get the rocket to sky, one has to apply RocketScience on it. For improvements of the old rockets, a scientist has defined a version 2.0. This includes a renaming from Rocket to Rocket20, the introduction of a new element for stability, that is the corpus. The engine has been

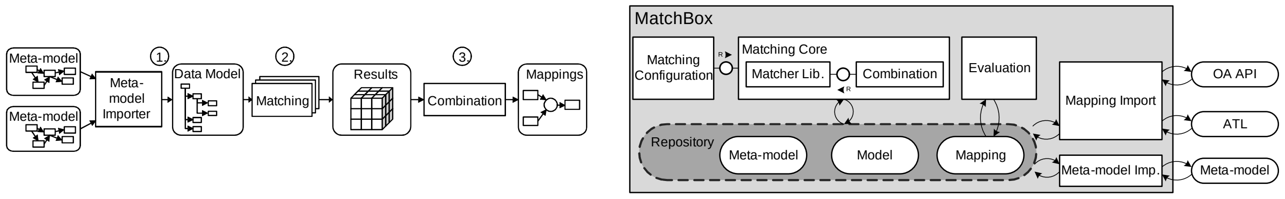


Figure 2: Processing and architecture of MatchBox

exchanged for an Ultimax2000, the NoseCone remains the same, but its role has changed to tip. Naturally, the new rocket has to be mounted on a MetalStick and can be only used by applying ImprovedRocketScience. A possible mapping would be to map the Rocket on Rocket20 and the other renamed elements to their counterparts, which is trivial but time-consuming. However, the newly introduced corpus has to be mapped too, that is a one to many mapping, i. e. a Rocket is mapped onto a Rocket20 and a Corpus, associated with its rocket.

In this example an automatic mapping simply based on name similarity of the elements in both meta-models will fail. However, as our results in the evaluation section show, an automatic mapping can be established when other matching algorithms are used. We have placed the partial output of a matching by MatchBox for this example in Fig. 1 (c).

4. MATCHBOX

In this section we will describe our matching system. Basically, it takes two meta-models as input and creates a mapping, i. e. correspondences between model elements, as output. In order to create mappings we propose to use a matcher framework that forms the base for combining results of different meta-model matching techniques. For this purpose, we adopted and extended a matcher combination approach as given in [16].

We have chosen the *SAP Auto Mapping Core* (AMC), that is an implementation inspired by COMA++ [4], a schema matching framework. In contrast, the AMC focuses on modularity, performance, adaptability and simplicity made of a set of matchers operating on trees, whereas COMA++ operates on a graph.

In the following we will explain MatchBox's architecture and its components. Afterwards, we outline the matching algorithms implemented demonstrating how they are applied to meta-models. Finally, we describe the combination of the different matchers' results by an aggregation and selection leading to a creation of mappings.

4.1 Processing Steps and Architecture

We adopt a traditional approach from schema matching by putting it into the context of MDA. It is build around an exchangeable matching core, enriching it with functionality for meta-model import, transformation import and evaluation. In order to create mapping results several steps have to be performed, as outlined in Fig. 2.

These steps are in detail: (1) The meta-models have to be transformed into the internal data model of the AMC. This step is necessary due to an adoption of schema matching techniques in AMC which are operating on trees. Having transformed the meta-models, several matchers can be applied each leading to separate results for two meta-model elements (2). The system can be configured by choosing which matcher should be involved in the matching process. Each matcher places its results into a matrix containing the similarity values for all source/target element combinations. These matrices are arranged in a cube, which needs to be aggregated in order to select the results for a mapping. This is done in the third step (3) to form an aggregation matrix (e. g. by calculating the average). Selection refers to a filtering of entries in the matrix of mappings, e. g. by selecting all elements exceeding a certain threshold. Finally, the selected entries are used to create a mapping.

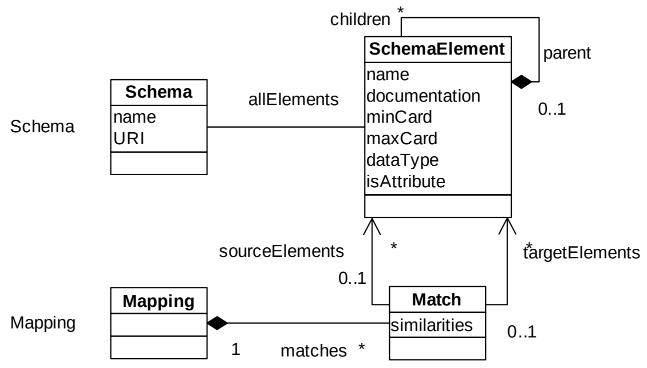


Figure 3: Meta-model of internal data model

Figure 1 shows an overview of MatchBox's architecture. As described before it uses an exchangeable matching core, which includes a set of matching components (matchers) operating on the information provided by an internal representation of meta-models. Additionally, it contains a combination component providing means for an aggregation and selection of results. The core, configuration and combination component are implemented by the AMC.

The matching core operates on the repository which contains all meta-models, provided models (instances) and mappings. The mapping importer implements an import from a transformation language (e.g. ATL [8]) or mapping language (e.g. Ontology Alignment API (OAA) [5]) to the mapping model of the AMC.

The configuration component implements the configuration of the aggregation and selection strategies. The meta-model importer transforms meta-models into the internal data model. The data model of the repository imposed by the AMC matchers is introduced in the subsequent section.

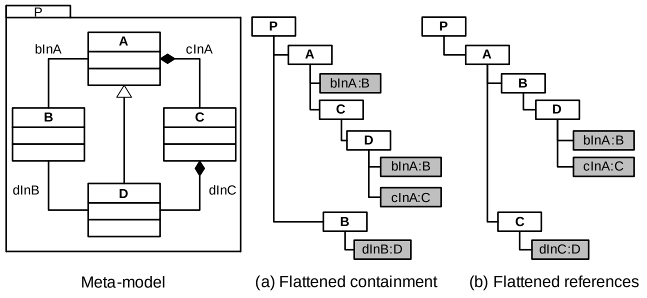


Figure 4: Examples for import approaches from a meta-model to the internal data model

4.2 Importing Meta-models into MatchBox

The internal data model is an acyclic connected graph where each node has one or more children and a maximum of one parent.

Figure 3 depicts the meta-model defining our internal data model consisting of a schema and schema elements. A schema acts as a container for a set of schema elements. Each schema element is related to a parent and contains a set of child schema elements. Furthermore, mappings are defined as a set of matches referencing the schema elements determined to be similiar. Each match contains a set of similarity values.

In order to process meta-models with MatchBox the first step is a conversion from a meta-model into the internal tree-based data model (schema). This includes a mapping from meta-model elements onto the schema. The mapping consists of two parts: (1) the element and (2) the relation mapping.

The element mapping (1) is defined element-wise; the global meta-model container is mapped onto a schema. A package, a class as well as an attribute, and reference are each mapped onto a schema element, setting its name accordingly. A schema element representing an attribute or reference is explicitly marked as an attribute, mapping its cardinalities as given in the meta-model. The parent of created attribute schema elements is set to the schema element representing the former parent class. Data types and enumerations are serialized using their names as types for the attributes.

The more challenging part is the mapping of the relations (2) between meta-model classes on a parent child hierarchy. The problem consists of a mapping from a graph on a tree. Several approaches to this problem can be found in literature. The closest one in mapping a meta-model onto a tree is the *Minimal Spanning Tree* (MST) of a meta-model. A MST is a sub graph of a graph connecting all nodes, thereby forming a tree with the sum of the costs of all edges being minimal. We opt for an established algorithm in order to determine the MST of a meta-model which is Kruskal's algorithm [10].

There are three types of relations: (1) References in general, (2) containment references, and (3) inheritance. The mapping to the tree structure starts with a package and the elements contained. In order to compute the MST first all references (1) or all containment references (2) are handled as edges of a graph. Then the classes of these references are handled as nodes and Kruskal's algorithm is applied. We added the additional constraint that a connection between elements is superior to the containment of the corresponding package. This ensures that the most simple approach which would be a root package containing all the classes is skipped. An example for a metamodel mapping is depicted in Fig. 4. In Fig. 4 (a) the resulting tree is formed by the flattened containment hierarchy, therefore C is contained in A, and D in C, whereas B follows the containment in the package P. The flattened reference import is depicted in Fig. 4 (b) having the path A, B, D, because the algorithm takes the left side path first (assuming these elements have been created first). Since we are calculating the MST, no element will have multiple occurrences in the resulting tree.

Besides the problem of defining a proper tree structure, inheritance has to be dealt with. Since the internal data model does not consider inheritance between elements, there are two approaches to overcome this problem. First, one just ignores the semantics of inheritance and discards this information. A second approach is to *flatten* the meta-model, i. e. copying all attributes and relations of a super class into its sub classes. The second preserves the information defined by inheritance, but loses the connection between super and sub classes. In Fig. 4 (a) and (b) show a flattened import based on the reference and containment hierarchy.

Applying this import transformation allows for meta-model matching by using different matchers on the internal representation. These matchers will be described in the following section.

4.3 Applying Matchers in MatchBox

The matchers of the core (AMC) operate on the internal data model (schema). Therefore, the information provided by the schema can be used for a determination of correspondences (matches). A matcher takes two schema elements as input and produces their similarity value as output.

Adopting the XML-schema based AMC, we applied a set of matchers implemented, namely they are: *name matcher*, *name path matcher*, *leaf matcher*, *parent matcher*, and *data type matcher*. We developed the following additional matchers for the given AMC: *children matcher*, *sibling matcher*, and *extended data type matcher*. The concepts of the matchers implemented are described in the following.

Name matcher. This matcher targets the linguistic similarity of meta-model elements. It splits given labels into tokens following a camel case approach. Afterwards, for each token a similarity based on trigrams is computed. The trigram approach determines the total count of equal character sequences of size three (trigram) and finally compares them to the overall number of trigrams.

Name path matcher. This matcher performs a name matching on the containment path of an element. Hence, it supports to distinguish sublevel-domains in a structured containment tree even if leaf nodes do have equal names. Essentially, a name path is a concatenation of all elements along a containment path for a specific element, e. g. the element Rocket (Fig. 5 (b)) has the name path *Metamodel/Rocket*. For matching the name matcher is applied on both name paths, thereby seperators are omitted and the similarities are combined.

Parent matcher. This matcher follows the rationale that having similar parents indicates a similarity of elements. The parent matcher computes the similarity of a source and target element by applying a specific matcher (e.g. the name matcher) to the source's and target's parents and returns

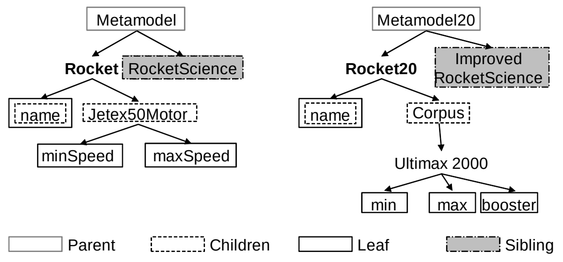


Figure 5: Examples for matching techniques

the similarity calculated. In Fig. 5, the example depicts two trees originating from the example as given in Fig. 1. Applying the parent matcher leads to a comparison of the parents *Metamodel* and *Metamodel20*, thereby we apply the leaf matcher. Finally, the values are aggregated as given in section 4.4 using the average strategy resulting in a value of 0.639.

Leaf matcher. This matcher computes a similarity based on similar leaf children. Thereby, the sub tree beneath an element is traversed and all elements without children (leaves) are collected. This set of leaves corresponding to a source element is compared to the set belonging to a target element by applying the name matcher and aggregating the single results for the source and target element. The example in Fig. 5 shows the elements considered when using the leaf matcher on *Rocket* and *Rocket20*. The leaf matcher combines the name and data type matcher while comparing all elements of both sets of leafs.

Data type matcher. The data type matcher uses a static data type conversion table. In contrast to the type system provided by XML, meta-models and in particular EMF allow a broader range of types. For example, EMF allows defining data types based on Java classes. We extended the data type matcher by conversion values for meta-model types and a comparison of instance classes. For instance, comparing two attributes one of type *EFloat* the other of type *EInt*, the data type matcher evaluates their datatypes, performs a look-up on its type table and returns a similarity of 0.6.

Children matcher. The children matcher follows the rationale that having similar child elements implies a similarity of the parent elements. This matcher uses any matcher to calculate an initial similarity, for the implementation we chose the leaf matcher. The children matcher evaluates the set of available children for a given source and target node. Comparing both sets by applying the leaf matcher leads to a set of similarities which are combined using the average strategy. Figure 1 shows an example for the children matcher. The children matcher uses the direct children of both elements leading to the result of 0.168.

Sibling matcher. The sibling matcher follows an approach similar to the children matcher. It is based on the idea that a source element S_s having siblings with a certain similarity to the siblings of a given target element S_t indicates a specific similarity of both elements. As in the children matcher, any matcher can be used for the calculation of similarity values for the siblings of S_s. In our implementation, we again chose the leaf matcher. The results of the separate matchings between a sibling and S_t's siblings are stored in a set. Finally, the set is combined as in the children matcher using the average of all values. The result is the sibling matcher's similarity for a given S_s and S_t. In Fig. 5 the sibling matcher is applied on *Rocket* and *Rocket20*, thereby comparing their siblings *RocketScience* and *ImprovedRocketScience*, combining the similarity leads a result of 0.7.

4.4 Combining Matcher Similarity Values

Following AMC, each matcher produces an output result in form of a matrix. This matrix orders the source and target elements along the x resp. y axis; the cells are filled with similarity values between 0 and 1. All similarity matrices are arranged along the matcher types (z axis) resulting in a cube.

In order to combine the results obtained, the combination component supports different strategies adopted from AMC like proposed in [1] or [4]. The strategies are *Aggregation*, *Selection*, *Direction* and a *Combination* of them. The aggregation reduces the similarity cube to a matrix, by aggregating all matcher result matrices into one. It is defined by four strategies: *Max*, *Min*, *Average*, and *Weighted*. The selection filters possible matches from the aggregated matrix according to a defined strategy. Possible strategies are *Threshold*, *MaxN*, and *MaxDelta*. The direction is dedicated to a ranking of matched schema elements according to their similarity. Matchers which use other matchers need to combine similarity values, e. g. the children and sibling matchers. This is covered by providing two strategies: *Average* and *Dice*.

These combination strategies grant the possibility of configuring the matching results by using the strategies and their parameters as outlined before. Typically, a default strategy is defined for a user of MatchBox. Following the matching process, having aggregated and selected the similarity values, the mappings are created.

5. EVALUATION

One of the questions to semi-automatic matching approaches is the relevance in terms of quality. How many mappings can be found, how many of them are correct and to which extent do they cover transformations implemented manually? We tackle these questions by providing an evaluation of MatchBox. This evaluation is based on 55 real-world transformations and 8 mappings, aggregated in our evaluation system. First, we will explain how we imported the mapping information from these transformations. Subsequently, we present our evaluation results using established statistical measures from the field of information retrieval [14]. Finally, we discuss our evaluation, its strengths and limitations as well as its consequences.

Our evaluation is based on a framework proposed by us, providing an automatic creation of so-called gold-standards. These gold-standards are then compared to the results obtained by matching. A gold-standard is a set of correct matches between meta-models, i. e. a set of mappings. Since a transformation is an implementation of a mapping, we select transformation definitions as gold-standards. The ATL-Zoo[1] provides a collection of 103 transformations of different sources and characteristics implemented in ATL[2]. It ranges from synthetic example transformations and integration transformations between technical spaces (TS) [11] to domain language and refactoring transformations. We omit-

[1]`http://www.eclipse.org/m2m/atl/atlTransformations/`
[2]`http://www.eclipse.org/m2m/atl`

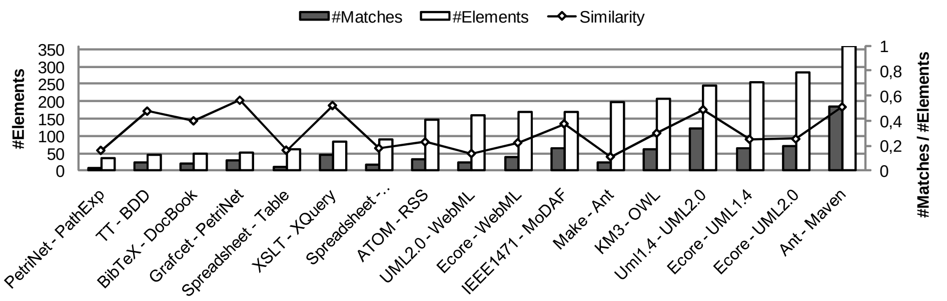


Figure 6: Dimension of 17 selected meta-models for matching

ted the refactoring transformations, since they are in-place transformations, thus the perfect solution from an automatic matching point-of-view is the identity transformation. We omitted the ones not based on EMF, because they can not be easily imported in our EMF-based system. Additionally, we also took the example mappings used in the evaluation of [17], because they consider evolution and large-sized example mappings, such as UML 1.4, UML 2.0 and Ecore.

In order to create the gold-standards needed for an evaluation, we first transform the ATL scripts into our mapping model. Establishing both the gold-standard and match originating mapping in one format allows for their comparison and evaluation. In our Java-based implementation, we first created an ATL model from its script, afterwards, we applied a model-to-model transformation, transforming the ATL model into our mapping model. This transformation is based on a depth-first search and implemented recursively, traversing all elements triggering the creation of mapped elements. We consider rules but only import conditional and sequential OCL-expressions. For the mappings of [17] we implemented an import from their format, the ontology alignment API (OAA) [5].

In order to describe the complexity of the mappings and participating meta-models we give an overview in Fig. 6. The diagram describes the complexity of the meta-models and their similarity. We show the number of matches between both meta-models mapped and the number of elements. The number of matches is the number of single mappings between elements, whereas the number of elements is the sum of all elements of both meta-models. Finally, the similarity is given by the match element ratio. In contrast to other meta-model matching approaches we provide a rather big range of mappings considered, ranging from 47 up to 360 elements, also the similarity varies from 0.21 up to 0.6.

Interestingly, smaller mapping examples such as *TT - BDD* have a higher similarity than bigger ones. This is due to the fact that mappings for large meta-models do not cover all elements, but rather a specific part, whereas in smaller meta-models most of the elements are mapped. There are three peaks for large meta-models, one at the *IEEE1471 - MoDAF* and at the *UML1.4 - UML2.0* and one at the *Ant - Maven* mapping. In the first case both meta-models actually describe the same domain, having only slight differences. In the second case, a mapping is defined between two versions of UML, leading to a high similarity, whereas in the last case, again two very similar domains are defined.

The measures that we used to evaluate the mappings' quality were precision, recall and F-Measure [14].

Precision p is the share of correct results relative to all results obtained by matching. One can say precision denotes the accuracy of the matching, for instance a precision of 0.8 means that 8 of 10 matches found are correct. *Recall r* is the share of correct found results relative to the number of all results (including the ones not found), i. e. a recall of 1 specifies, that all mappings were found, however, there is no statement how much more (incorrect) were found. Precision and recall influence each other which means an increase in recall leads to a decrease in precision and vice versa. Having a high precision leads to a few correct matches, thus reducing the overall number. Therefore, neither precision nor recall alone allow for a statement. *F-Measure F* represents the balance between precision and recall, it is commonly used in the field of information retrieval and applied by many matching approach evaluations, e. g. [7, 4, 13].

Let t_p be the true positives, i. e. correct results found, f_p the false positives, i. e. the found but incorrect results, and f_n the false negatives, i. e. not found but correct results, then the formulas for these measures are:

$$p = \frac{t_p}{t_p + f_p},\ r = \frac{t_p}{t_p + f_n},\ F = 2 \cdot \frac{p \cdot r}{p + r}.$$

The F-Measure can be seen as the effectiveness of the matching balancing precision and recall equally. For instance, a precision and recall of 0.5 leads to a F-Measure of 0.5 stating that half of all correct results were found and half of all results found are correct.

5.1 Results

We performed our evaluation with respect to the goal of determining our system's quality in terms of found mappings and also to evaluate the parameters described in Sect. 4.2. Therefore, we considered the following constellations: (1) Containment-based vs. reference-based import, (2) flattening vs. non-flattening, (3) best configuration of matchers vs. an average configuration. This leads to eight possible combinations and series of values to be measured. Additionally, we investigated the occurence of matchers in combinations of 5 and 6 matchers out of the 7 available.

First, we performed tests for all combinations of aggregation strategies, selection directions and selection strategies for each transformation. Based on first results, also confirmed later on by our matcher combination evaluation, we omitted the data type matcher and chose the name, name path, children, sibling, parent and leaf matchers. We performed tests with selection of the maximal to the five best results (*MaxN*), changing the *Delta* value to range between 1 and 10% to find the best match and any other matches that fall within the defined range. The selection based on *Threshold* was performed for ranges from 0.3 to 0.9.

All parameter configurations were combined with the possible types of meta-model imports (containment, reference and flattening). The execution of these tests produced 2,268 results for each sample ATL code (55) and OAA mapping

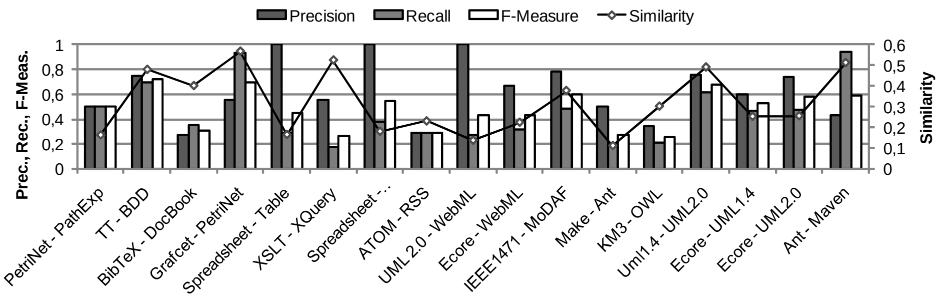


Figure 7: Selection of 17 results with best configuration

(8), giving overall 142,884 results for all 63 examples.

Having a closer look, we decided to omit 40 of the transformations. We realized that these transformations were either integration transformations between technical spaces (23), e. g. a transformation from a meta-model to XML by defining a schema (e. g. *R2ML - XML*, *RDM - XML*). This implies a mapping from a typed model into a schema definition were most of the source elements are mapped to the same target element, setting the target's type using constant expressions.

We also omitted the synthetic examples (17), such as *Families - Persons*. In the following, we refer to the remaining 15 transformations and 8 OAA mappings while discussing our results. We investigated the configurations of the different 23 results and determined the best configuration in terms of F-Measure for each of them.

We depict a selection of results for specific transformations with flattened containment-based import in Fig. 7. We also show the similarity, to relate it to our results. The similarity and recall obtained show a certain proportional relation which is a common observation in information retrieval, the more similar meta-models are, the more matches can be found.

However, there are two execeptions, *BibTeX - DockBook* and *XSTL - Xquery*. This grounds in the kind of mapping, because it contains several constant expressions, i. e. an element is mapped on a certain constant attribute value, e. g. a specific string value. These mappings are impossible to detect, using our current matching techniques.

The overall maximal F-Measure that was achieved in our tests is 0.72 for the transformation *TT - BDD*, i. e. 72% of the mappings are created automatic, however for the more complex example *UML1.4 - UML2.0*, we achieved 0.65. The maximal recall and precision results were 0.95 and 1. The average for precision, recall and F-Measure in flattened containment is 0.61, 0.47 and 0.53 respectively. For flattened reference they are 0.50, 0.41 and 0.45, the non-flattened containment yields 0.58, 0.50 and 0.54, where non-flattened reference gives 0.51, 0.52, and 0.51. The results are depicted in Fig. 8.

Based on the maximal results for each example, we determined an average of all parameters to create an average configuration for the matchers. The average configuration is: the *weighted* aggregation strategy with the weighted value for each matcher being calculated as the average from the corresponding weight in the maximum (best) results. The best selection direction was *Both* and the selection strategy *Multiple* where the *Delta* parameter was set to 0.04 and the *Threshold* to 0.3. We performed tests with the average configuration for all different types of imports. The values obtained are depicted in Fig. 8.

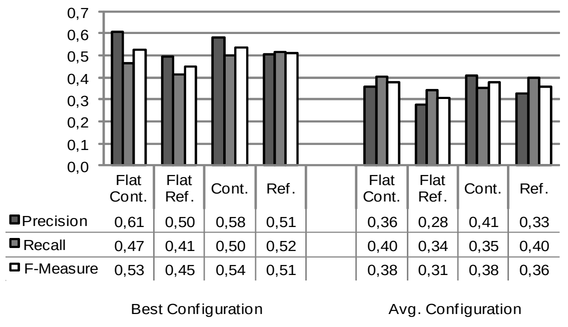
Last, we tested the effect of different matcher combinations on the final results. Based on the flattened containment import with average configuration, we selected all 5 and 6 matcher combinations out of the 7 available. We took the best results for F-Measure and investigated the occurrence count of each matcher used.

5.2 Discussion

Our evaluation has shown that MatchBox leads to an average of 54% mappings found. Therefore, our system constitutes a suitable approach for semi-automatic meta-model matching. The best result is achieved by a non-flattened containment-based import using our best combination configurations. This leads to a second insight; using containment is slightly better than flattened containment and superior to reference import. This is due to the fact that in the context of meta-model matching, a tree based on a flattened containment approach captures more of a meta-model's information than the other approaches. We expected the flattening to perform better. But due to the fact that our transformation import does not consider the additional mappings resulting from the flattening, e. g. imagine the *NamedElement* and its inheritance, the result gets worse. Therefore, future work will cover an investigation how to handle inheritance efficiently.

Observing the matchers used, we realized that matchers with structural information, i. e. children and sibling matcher are part of the best performing configurations. Therefore, the structural information constitutes an important part in matching meta-models. A fourth statement can be made; the improvement gained by using the best configuration is on average 30% compared to an average configuration. It can be concluded that an algorithm or classification of meta-models to be matched and a corresponding configuration yields improvements in the range mentioned. Finally, we noted through our evaluation that meta-model matching cannot be applied in most cases of transformations covering technical space integration.

The strength of our evaluation lies in an automatized way of evaluating matching approaches and the broad range of reference mappings available by importing several transformations from the ATL-Zoo. However, the limitations of our evaluation are the limited capabilities of our ATL-Importer, not taking complex OCL-expressions into account. Additionally, our test data comprises 23 transformations which may not cover the full spectrum of model transformations and needs an extension with additional transformations. In particular, the most suited examples, the model evolution (versioning) transformations, are missing, although they generally lead to very good results. We considered 4 examples for this, but an extension with more mappings should be done in the future. We are also aware of a missing compar-



	Flat Cont.	Flat Ref.	Cont.	Ref.		Flat Cont.	Flat Ref.	Cont.	Ref.
▪ Precision	0,61	0,50	0,58	0,51		0,36	0,28	0,41	0,33
▪ Recall	0,47	0,41	0,50	0,52		0,40	0,34	0,35	0,40
□ F-Measure	0,53	0,45	0,54	0,51		0,38	0,31	0,38	0,36

Figure 8: Results for flattened, containment and reference combination import

ison to other matching approaches. This could be part of further work, eventually leading to a dedicated automatic benchmark for meta-model matching, e. g. by adopting the ontology alignment API [5].

6. CONCLUSIONS

Meta-model matching is a promising field having recently received interest. With MatchBox, we presented our approach on meta-model matching, namely a framework that facilitates a combination of matchers to achieve the best results. We described how we adopted schema matching techniques to MDA and the problems encountered. Moreover, solutions to import a meta-model into a tree-based model were given and an application of matchers to the resulting models was outlined as well as a subsequent combination of matcher results. For an evaluation of our results we proposed a benchmark architecture using 23 existing model transformations and mappings. We implemented MatchBox and performed an evaluation according to our proposals.

The contributions of our work are: (1) A generic matching combination framework, (2) an evaluation of an adoption of existing schema matching techniques, (3) a meta-model to tree transformation in context of matching, (4) the definition of a benchmark using model transformations for evaluation of matching solutions

Our prototypical implementation and the adoption of the AMC have shown that meta-model matching yields promising results; i. e. up to 72% mappings found (54% in average). Thereby, we have shown that a transformation of a meta-model to a tree structure based on a containment hierarchy yields the best results for matching, since it preserves most of a meta-model's semantics. Our evaluation of schema matching techniques has shown that the ones also using structural information perform best. We can further conclude that our benchmark constitutes a base for an evaluation of any meta-model matching approach.

Nevertheless, the benchmark leaves room for improvement regarding the coverage of transformation domains and should be extended by additional transformations. The import of ATL-rules with respect to complex OCL-expression also needs an additional refinement, thus capturing all mapping information. The approach of automatic extraction of gold-standards from existing transformations can be extended to data bases as well as ontologies. For instance, applied on query languages, corresponding gold-standards can be automatically created. Additionally, the meta-model transformation to a tree results in a loss of information. Therefore, our next step is to define a generic data model also capturing information like inheritance, graph structures, instances, etc. Since our evaluation has shown the effectiveness of structural matchers, we will develop dedicated graph matchers as well as matchers taking our data model's and instance characteristics into account.

7. REFERENCES

[1] P. A. Bernstein, S. Melnik, M. Petropoulos, and C. Quix. Industrial-strength schema matching. *SIGMOD Rec.*, 33(4):38–43, 2004.

[2] N. Choi, I.-Y. Song, and H. Han. A survey on ontology mapping. *SIGMOD Rec.*, 35(3):34–41, 2006.

[3] K. Czarnecki and S. Helsen. Feature-Based Survey of Model Transformation Approaches. *IBM Systems Journal, Special Issue on Model-Driven Software Development*, 45(3):621–645, July 2006.

[4] H. H. Do. *Schema Matching and Mapping-based Data Integration.* VDM Verlag Dr. Mueller e.K., 2006.

[5] J. Euzenat. An API for ontology alignment. In *Proc. of ISWC'04*, pages 698–712, 2004.

[6] M. D. D. Fabro and P. Valduriez. Semi-automatic model integration using matching transformations and weaving models. *Proc. of SAC '07*, pages 963–970, 2007.

[7] J.-R. Falleri, M. Huchard, M. Lafourcade, and C. Nebut. Metamodel matching for automatic model transformation generation. In *Proc. of MoDELS '08*, pages 326–340, 2008.

[8] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Sci. Comput. Program.*, 72(1-2):31–39, 2008.

[9] G. Kappel, H. Kargl, G. Kramler, A. Schauerhuber, M. Seidl, M. Strommer, and M. Wimmer. Matching Metamodels with Semantic Systems – An Experience Report. In *Proc. of BTW 2007*, 2007.

[10] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. In *Proc. of the American Mathematical Society*, volume 7, page pp. 48–50, 1956.

[11] I. Kurtev, J. Bézivin, and M. Aksit. Technological Spaces: an Initial Appraisal. In *International Federated Conference, Industrial Track*, 2002.

[12] D. Lopes, S. Hammoudi, and Z. Abdelouahab. Schema matching in the context of model driven engineering: From theory to practice. In *Proc. of SCSS'05*, pages 219–227, 2006.

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

[14] C. J. V. Rijsbergen. *Information Retrieval.* Butterworth-Heinemann, 1979.

[15] P. Shvaiko and J. Euzenat. A survey of schema-based matching approaches. *Journal on Data Semantics*, 4:146–171, 2005.

[16] K. Voigt. Towards combining model matchers for transformation development. In *Proc. of FTMDD'09*, 2009.

[17] M. Wimmer and G. Kramler. Bridging grammarware and modelware. In *MoDELS Satellite Events*, pages 159–168, 2005.