On Realizing a Framework for Self-Tuning Mappings*

Manuel Wimmer, Martina Seidl, Petra Brosch**,
Horst Kargl, and Gerti Kappel

Vienna University of Technology, Austria
lastname@big.tuwien.ac.at

Abstract. Realizing information exchange is a frequently recurring challenge in nearly every domain of computer science. Although languages, formalisms, and storage formats may differ in various engineering areas, the common task is bridging schema heterogeneities in order to transform their instances. Hence, a generic solution for realizing information exchange is needed. Conventional techniques often fail, because alignments found by matching tools cannot be executed automatically by transformation tools. In this paper we present the Smart Matching approach, a successful combination of matching techniques and transformation techniques, extended with self-tuning capabilities. With the Smart Matching approach, complete and correct executable mappings are found in a test-driven manner.

1 Introduction

In this paper we present a self-tuning approach for information integration. Our approach—the Smart Matching approach—allows the derivation of high quality executable mappings for different kinds of schemas from small, simple examples defined by the user.

Seamless information exchange between different sources is an important task in nearly every engineering area of computer science. This ubiquitous challenge recurs in various application domains starting from the exchange of data between databases over the exchange of ontology instances between semantic web services to the exchange of complete models between modeling tools. Although every engineering area has its own languages, formalisms, and storage formats, the problem is always the same. Two heterogeneous schemas have to be bridged in order to transform instances of the one schema to instances of the other. Thus, a generic information integration solution for the use in different application domains is highly valuable.

* This work has been partly funded by FFG under grant FIT-IT-819584 and FWF under grant P21374-N13.

** Funding for this research was provided by the fFORTE WIT - Women in Technology Program of the Vienna University of Technology, and the Austrian Federal Ministry of Science and Research.

On the one hand, matching techniques for automatically finding semantic correspondences between schemas have been developed in the context of information integration. On the other hand, transformation techniques have been established for automatically transforming data conforming to a schema A to data conforming to a schema B in a semantic preserving way. However, the combination of matching approaches and transformation approaches is accompanied with several difficulties due to the huge gap between detected alignments, mappings, and executable transformation code. For bridging this gap, two problems have to be solved. First, state-of-the-art matching techniques do not produce executable transformation code. Second, for the automatic execution of the transformation, correctness and completeness of the found alignments are indispensable. A solution for the first problem is presented in [7], where reusable mapping operators for the automatic transformation of models are introduced. A solution for the second problem is conceptually proposed in [8], where we discussed how to improve current integration techniques by the application of self-tuning methods in a test-driven manner resulting in the Smart Matching approach. Based on this work, we implemented the self-tuning matching framework *Smart Matcher*.

In this paper we describe how the Smart Matcher increases the completeness and correctness of alignments based on predefined examples. In particular, we present the realization of the *Fitness Function* for self-evaluation and the *Mapping Engine* for self-adaptation which are the basis of the self-tuning.

This paper is organized as follows. In Section 2 we present the Smart Matching approach at a glance and the running example of this paper. Sections 3 and 4 elaborate on the two core components of the Smart Matcher namely the Fitness Function and the Mapping Engine, respectively. In Section 5 we present an evaluation of our approach. Related work is discussed in Section 6. In Section 7 we conclude and give an outlook to future work.

2 Smart Matching at a Glance

In this section, we present the Smart Matcher and its underlying conceptual architecture at a glance. The Smart Matching method combines existing integration techniques, i.e., matching and transformation approaches, by introducing a dedicated mapping layer for bridging the gap between alignments, mappings, and executable transformation code as described in [1]. Feedback-driven self-tuning improves the quality of the mappings in an iterative way [9]. For the implementation of the self-tuning capabilities, the quality of mappings has to be evaluated. Therefore, we adopt test-driven development methods from software engineering [3] for developing integration solutions. The Smart Matcher's architecture is illustrated in Fig. 1. Basically, the application of the Smart Matcher comprises eight phases which are described in the following.

1. Develop example instances. Like in test-driven software development, we specify an expected outcome for a specific input, i.e., instances for input and output

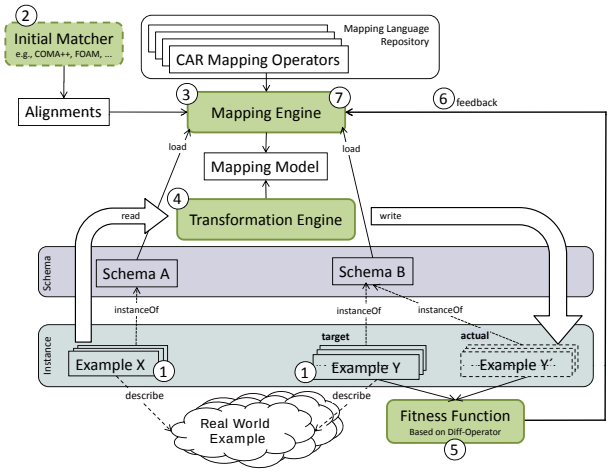

Fig. 1. The Smart Matching approach at a glance.

schemas representing the same real world example are developed. Those instances are necessary to verify the mappings identified in later phases in order to follow a feedback-driven integration method. The user defined input instance is later called *LHS (left-hand side) instance*, the output instance is named *target RHS (right-hand side) instance*. This is the only time during the whole Smart Matching process where human input is necessary.

2. Generate initial alignments. Existing matching tools create initial alignments which may serve as input for the Smart Matcher. Every tool which supports the INRIA alignment format [4] may be used. This phase is optional, i.e., the Smart Matcher is also able to start from an empty set of alignments.

3. Interpret initial alignments. The alignments produced in Phase 2 are transformed into an initial mapping model. This task is performed by the Mapping Engine.

4. Transform instances. In this phase, the Transformation Engine executes the initial mapping model and thereby transforms LHS instances into actual RHS instances.

5. Calculate differences. The Fitness Function compares the actual and the target RHS instances by means of their contained objects, values, and links. The differences between the instances are collected in a `Diff-Model`, which expresses the quality of the mappings between the source schema and the target schema.

6. Propagate differences. The differences, i.e., missing and wrong objects, values, and links, identified by the Fitness Function are propagated back to the Mapping Engine. The feedback is based on the assumption that schema elements are not appropriately mapped if differences are calculated for their instances.

7. Interpret differences and adjust the mapping model. The Mapping Engine analyzes the feedback and adapts the mapping model between source and target schema by searching for and applying appropriate mapping operators. Depending on the types of schemas to be integrated and the used mapping language, different kinds of mapping strategies may be used.

8. Iteration. Now Phase 4 and Phase 5 are repeated. If the comparison in Phase 5 does not detect any differences or if a certain threshold value is reached, the process stops. Otherwise, the Smart Matcher's Mapping Engine adapts the mapping model based on the identified differences and starts a new iteration by returning to Phase 4.

In the following, we elaborate on the *Fitness Function* necessary for self-evaluation in Phase 5 and the *Mapping Engine* necessary in particular for the self-adaptation in Phase 7. Therefore, the next sections comprise detailed descriptions on the design rationale and implementation of these two components based on the example depicted in Fig. 2 (a). The schema on the LHS assigns a `discount` with a `height` to each `customer` who is characterized by `name` and `famName`. The schema on the RHS describes similar information, namely the `rebate height` of a `person`. Only the family a person belongs to is modeled by its own class. A correct mapping between those schemas should detect that the attribute `famName` ought to be mapped to the class `family` what is not given by the mapping shown in Fig. 2 (b) which has been produced from the automatically computed alignments (cf. Fig. 2 (a)).

3 A Fitness Function for Mapping Models

The aim of the Fitness Function is to provide feedback on the quality of the mapping models to the Mapping Engine. In the transformation scenario, the quality of a mapping model indicates the correctness of the transformation of the input model to the corresponding output model. Using a test-driven approach where input and output pairs (e.g., cf. Fig. 2 (c)) are given, the generated output model is compared with a given target output model (cf. Fig. 2 (d)). The smaller the differences between the actual output model and the target output model are, the better the quality of the mapping model is. This raises the question how to compute expressive differences between actual and target instance models. When using object-oriented schemas which typically consist of classes, attributes, and references, differences on the instance level appear between objects, values, and links. In the following, we describe our heuristic-based comparison method which is sufficient to compute the necessary feedback for the mapping engine without applying expensive comparison algorithms.

Comparing Objects. In general, objects have a unique ID within an instance model (cf. Fig. 2 (c), e.g., `f1:Family`). However, when we want to compare actual instance models with target instance models which are independently created, one may not assume that two objects with the same ID are equivalent. This is because often IDs are arbitrarily assigned, e.g., based on the order the

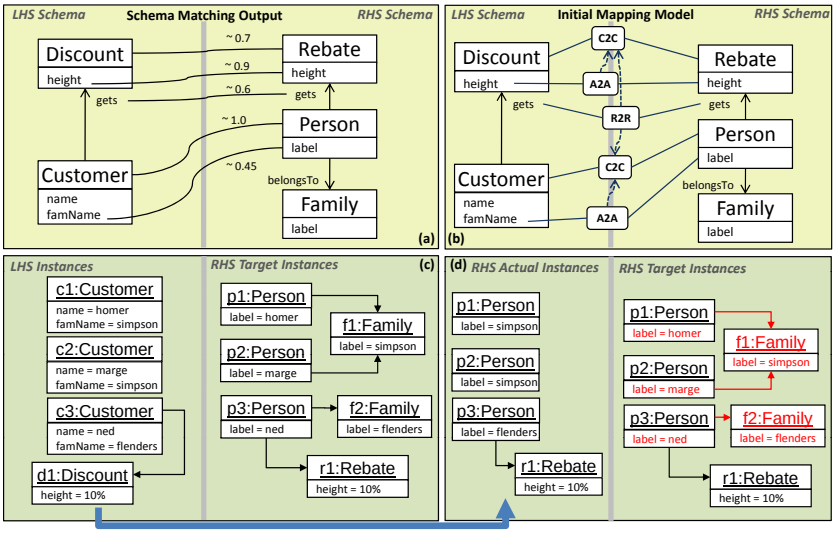

Fig. 2. Running example: (a) alignments, (b) initial mapping model (c) user-defined test instances, (d) actual vs. target instances for initial mapping model.

objects are instantiated. Therefore, we cannot rely on exact ID-based comparison approaches, instead we have to apply a heuristic. The weakest condition, which is necessary but not sufficient, for verifying that two instance models are equivalent is to count the objects of a particular class in each instance model and compare the resulting numbers, i.e., compute the cardinalities for a given class. Consequently, having the same amount of objects for each class of the RHS schema (abbr. with rhsSchema) in the actual instance model (abbr. with actual) and in the target model (abbr. with target), is an indication that the applied mappings between the classes of the schemas ought to be correct and complete. This consideration leads to the following condition.

```
Condition 1 := forall classes c in rhsSchema |
               actual.allObjects(c).size() = target.allObjects(c).size()
```

Because Condition 1 is not sufficient, it is possible that in certain cases the amount of objects is the same for a given class but the mappings are not correctly defined. Assume that for example a LHS class has a mapping to a RHS class, which is instantiated twice, but actually the LHS class should be mapped to another RHS class which is also instantiated twice. However, due to the fact that the RHS classes have both the same amount of instances, no difference can be determined when considering Condition 1 only. Thus, the mapping between the classes are falsely interpreted as correct. To be sure that two objects are equal, a deep comparison is necessary, meaning that attribute values and links of the objects under consideration have to match. Using deep comparison, this kind of false mappings between classes can be detected.

Example. Fig. 2 (d) illustrates on the LHS the actual instances generated by the interpretation of the automatically computed alignments and on the RHS the given target instances. When Condition 1 is evaluated on these two models, the cardinalities of the classes `Rebate` and `Person` are the same for the actual model and for the target model. However, the cardinality of the class `Family` is obviously not equivalent for both instance models. The `Family` objects are totally missing in the actual model.

Comparing Values. Having the same cardinalities for a specific class is a prerequisite for verifying that actual and target instance models are the same. To be sure that two objects, one in the actual and the other in the target instance model, are equal, these objects must contain exactly the same attribute values. Therefore, for each attribute of the RHS schema two sets of values are computed. The first one comprises all values of a specific attribute for the actual model and the second one does the same for the target model. In case these two sets comprise the same elements, the mapping for the RHS attribute seems to be correct. Otherwise the attribute mapping is assumed to be incorrect or missing.

```
Condition 2 := forall attributes a in rhsSchema |
               actual.allValues(a) = target.allValues(a)
```

Example. When Condition 2 is evaluated for our running example, one can easily see that for the attribute `Rebate.height` Condition 2 holds. However, for the attribute `Person.label` the computed sets are totally different, namely for the actual instances the resulting set is {`simpson, simpson, flenders`} which has no match with the computed set for the target instances {`homer, marge, ned`}. Furthermore, Condition 2 does not hold for `Family.label`, because in the actual model there are no `Family` objects that may hold such values.

Comparing Links. The last indication that the actual and target instance models are equal, and consequently that the mapping model between the schemas is correct, is that the structure of the instance models is the same, i.e., the links between objects must be equivalent. In particular, equal objects must have the same amount of incoming and outgoing links, which is in general hard and expensive to prove. In order to provide a fast comparison method for links, we decided only to check if the references of the RHS schema have the same amount of instantiated links.

```
Condition 3 := forall references r in rhsSchema |
               actual.allLinks(r) = target.allLinks(r)
```

Example. Considering our running example, Condition 3 holds for the reference `Person_gets_Rebate`. However, for the reference `Person_memberOf_Family` Condition 3 does not hold. In the actual model we have no instantiations of this reference, thus the result of the `allLinks` operation is the empty set, whereas in the target model the operation `allLinks` produces a set with three entries.

Preparing the feedback for the mapping engine. After evaluating these three conditions for all RHS schema elements, the feedback for the Mapping Engine is defined as the set of all RHS schema elements which do not fulfill

the aforementioned conditions. For the adaptation of the mapping model done by the Mapping Engine, we decided that it is currently sufficient to know the schema elements for which differences on the instance level have been found. Therefore, these schema elements are collected for the feedback only, and not the actual differences, i.e., objects, values, and links. Concerning our running example, the following set is computed as feedback for the Mapping Engine {`Family, Person.label, Family.label, Person_partOf_Family`}.

4 A Feedback-aware Mapping Engine

After the Fitness Function has computed the feedback, the Mapping Engine interprets it and adapts the current mapping model to improve its quality. In order to adapt mapping models in an appropriate and efficient way, we make use of two kinds of strategies. First, *local strategies* are needed to find out if a given mapping operator may be applied appropriately for a set of given schema elements. Second, a *global strategy* is needed to guide the mapping process in general and in particular for orchestrating the local strategies.

4.1 Local Strategies

As depicted in the Smart Matcher architecture (cf. Fig. 1), the LHS instance models are transformed into RHS actual instance models. Subsequently, the difference between the actual and target instance models is computed on the RHS which is then propagated back to the Mapping Engine. Consequently, the Smart Matcher always transforms form left to right but adapts the mapping models from right to left. For adaptation purposes, the Mapping Engine has to apply new mappings which can also replace existing ones. However, before a new mapping is applied, it has to be ensured that the selected mapping operator is applicable for a given set of schema elements, i.e., the mapping model should be executable and the execution should result in consistent RHS instances. In particular, each mapping operator needs a specific LHS structure and RHS structure as well as already applied mapping operators as context. Based on these constraints, local strategies can be derived for determining if a given mapping operator can be applied in a particular situation. Therefore, we have extended our mapping operators with a `isApplicable` method which checks for given LHS and RHS schema elements if a mapping can be applied in the current state of the mapping model. For determining the implementation of the `isApplicable` methods, information inherent in the mapping language is needed. This means, the local strategies can be derived from the abstract syntax and static semantic constraints of the mapping operators which have to be assured to generate correct and executable mappings. If the `isApplicable` method returns true for given LHS and RHS elements, the mapping operator can be applied. Thus, with the help of the local strategies, we are able to automatically build mappings between the schemas. However, until now, we have not defined how a mapping

model is actually build. Therefore, in addition to the local strategies, a global strategy is needed which guides the overall mapping process.

4.2 Global Strategies

Before a global strategy can be derived, one first has to think about how mapping models are manually developed. Generally, mapping models are developed in a three-step process. First, some prototypical mappings are established. Second, these mappings have to be interpreted, i.e., in the transformation scenario the RHS instances are generated from the LHS instances. Third, the output of the transformation has to be verified in order to be sure that the mappings were correctly developed. For automation purposes, these steps have to be covered by the global strategy. However, what aggravates the derivation of the global strategy is that there a several variations how to proceed in each step. For example, one user may prefer to build several mappings between classes in the first place before mappings between attributes and references are created. Whereas, another user may prefer to find only one mapping between two classes and then tries to find all attribute mappings for these two classes. These examples already show that the derivation of a global strategy is not as predetermined as the derivation of the local strategies. In particular, the following variation points for global strategies exist.

- **Sequence of mapping operator applications.** The first decision that has to be made is the sequence in which the different types of mapping operators are applied. One possibility may be first finding structural commonalities between the schemas by applying symmetric operators, followed by bridging structural heterogeneities by applying asymmetric operators.
- **Depth-first vs. breadth-first search**. Additionally to the application sequence of mapping operators, the global strategy may decide to find all C2C mappings first and afterwards A2A and R2R mappings are explored (*breadth-first*) or find iteratively one C2C mapping and for this mapping all possible A2A and R2R mappings are searched (*depth-first*).
- **All-at-once vs. incremental evaluation.** Finally, it has to be decided, when and how often mappings are evaluated. The global strategy may find several possible mappings and evaluate them all together (*all-at-once*) or evaluate each found mapping individually (*incremental*). The all-at-once approach is similar to the waterfall model in software engineering, with its disadvantage of spending much effort when working in the wrong direction. This can be avoided with the incremental approach; however, much more evaluation steps are necessary.

Because there are different ways to define a global strategy, we decided to use a state-machine based implementation approach for global strategies. The major advantage of this approach is to separate basic functionality, such as transforming models, computing differences between models, as well as searching and applying mappings, from the global strategy guiding the overall mapping

process. This allows us to reconfigure the global strategy faster than compared to reconfigurations on source code level, which is especially needed to empirically analyze various configurations of the aforementioned variation points. In the following, we present the most efficient configuration explored in our empirical experiments. Fig. 3 illustrates this global strategy as an UML state diagram, which is a depth-first and incremental strategy regarding to the aforementioned variation points. The depth-first search also determines the application sequence of the mapping operators. Please note that only those parts of the global strategy are illustrated which are relevant for our running example. The first task of the global strategy is to calculate the not yet correctly mapped LHS and RHS schema elements based on the output of the Fitness Function. Then, it proceeds with the establishment of mappings able to act as context mappings for other mappings. More specifically, the idea is to examine one mapping between a RHS class and an arbitrary LHS element, and then attributes and references of this RHS class are used to approve the correctness of this context mapping. This means, if a mapping for a RHS class is found and this mapping fulfills the cardinality constraint (cf. Condition 1 in Section 3), in the next steps, mappings for its features, i.e., attributes and references, are searched. If at least one feature mapping is evaluated to true, i.e., fulfills Condition 2 or 3 of Section 3, the context mapping is approved. For the opposite case, the context mapping is marked as false, although the cardinality constraints are fulfilled. After giving an overview on the design rational of this strategy, we discuss the three major phases of this strategy based on the state machine illustrated in Fig. 3.

Finding context mappings. As already mentioned before, the local strategies search from right to left. Hence, after the `Init` state where not yet correctly mapped LHS and RHS schema elements are computed, a RHS class is selected in the state `Select RHS Class` to establish a context mapping. The first choice is to select a LHS class to apply the C2C mapping operator (cf. state `Search 4 LHS Class`). If a C2C mapping has been found by the local strategy (cf. `isApplicable` method), it is applied and the state `Evaluation` is entered. Otherwise, the RHS class is stored in the C2C blacklist which has as impact that this RHS class will never be applied to a LHS class in future iterations (cf. condition `[not in C2C_BL]`).

Evaluating context mappings. Within the state `Evaluate`, the LHS instance model is transformed into a RHS actual instance model. After the transformation, the actual and the target instance models are compared and the output of the Fitness Function is analyzed. If the applied mapping is evaluated to true, e.g., for C2C mappings it is required that the cardinality constraint is fulfilled, consists of searching for feature mappings for the selected RHS class. In case the evaluation returned false, the RHS class and the LHS element are stored in the blacklist (i.e., this combination will never be applied again) and it is switched into the state `Init`.

Approving context mappings. When the context mapping is evaluated to true, the current state is changed to `Select RHS Att` in which a not yet correctly mapped attribute of the RHS class is selected. Subsequently, the strategy looks

for free LHS attributes. The structure and behavior of the following states for the application of attribute mappings is analogous to the aforementioned application of context mappings. After the evaluation of the found A2A mapping (cf. state `Evaluation`), three cases have to be distinguished. First, the A2A mapping is evaluated to true and the RHS class has no references, which results in an approved context mapping, i.e., it is not changeable in future iterations, and in a transition to the `Init` state. Second, no A2A mapping has been evaluated to true and the RHS class has no references which results also in a transition to the `Init` state but the context mapping is marked as evaluated to false, i.e., it is changeable in future iterations. Third, the RHS class has additional references, then the state is changed to `Select RHS Ref` independent of the evaluation of the A2A mapping. The mapping of the RHS references and approving of the context mapping is analogous to that of attributes.

Running Example. For exemplifying the execution behavior of the presented global strategy, we make use of the running example. In Section 3, we have elaborated on the output of the Fitness Function for the initial mapping

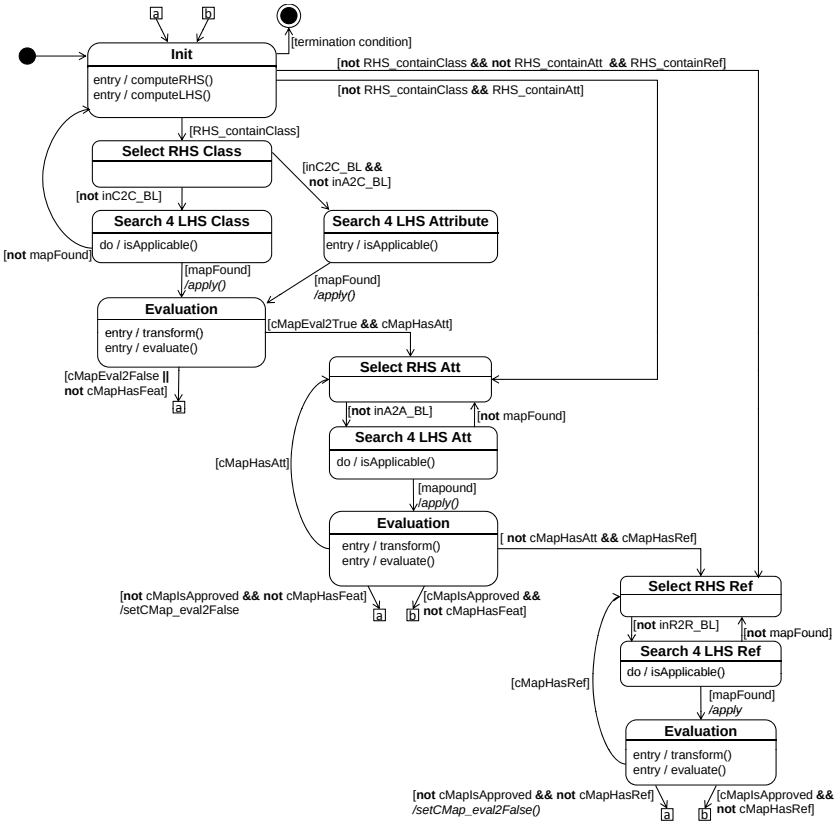

Fig. 3. Incremental depth-first global strategy implemented as UML state machine.

model which is summarized in Iteration 0 of Fig. 4. Now, it is explained how the global strategy adapts the mapping model based on this input. In addition, the necessary iterations are illustrated in Fig 4.

Iteration 1 and 2. When we reconsider the computed output of the Fitness Function, we find out that only one RHS class is not correctly mapped. In particular, the class `Family` has currently no mapping to a LHS element. Therefore, the global strategy switches from the `Init` state to the state `Select RHS class` in which the `Family` class is selected. On the LHS schema no free classes are available now. Thus, it is not possible to apply a C2C mapping and the global strategy searches for a free attribute on the LHS to apply a A2C mapping without aggregation. The `Customer.name` attribute is selected and an A2C mapping is applied. Subsequently, the `Evaluation` state is entered, the transformation is started, and the produced actual model is compared with the given target model. The evaluation results amongst others into differences for `Family` instances which is an indicator that the applied mapping is not correct. Thus, the found mapping is put onto the blacklist and the `Init` state is entered. Because there is another configuration possibility for the A2C operator, in iteration 2, an A2C mapping with aggregation is applied which is also not correct and therefore we end up again in the `Init` state.

Iteration 3. The `Init` state computes the same unmapped RHS and LHS elements as in the previous iterations. Thus, we enter again the `Select RHS class` state where the class `Family` is selected. Again, no class on the LHS is free and a LHS attribute is selected. In this iteration, we already know that `Customer.name` is not the corresponding element. Therefore, another free attribute is searched which results in selecting `Customer.famName`. After building the A2C mapping marked without aggregation between `Family` and `Customer.famName`, the `Evaluation` state is entered. This time, the evaluation shows that the cardinalities of the class `Family` are not equal, but some attribute values overlap. However, for evaluating this mapping to true, the similarity value is too low. Therefore, the next iteration is started by switching to the `Init` state.

Iteration 4. After entering the `Init` state, we have again the same set of incorrectly mapped RHS elements. Thus, the same path is followed as before. But this time, the applied A2C mapping is marked with aggregation. This means, only for unique values new objects are created. The evaluation of this mapping shows that no differences may be found for the focused RHS elements and the mapping is marked as evaluated to true. Because the `Family` class has no further unmapped features, the state `Init` is entered.

Iteration 5. This time, the `Init` state computes that only one RHS attribute is not appropriately mapped. Therefore, we directly navigate to the `Select RHS Att` state in which `Person.label` is selected. Within the next state, `Customer.name` is selected, because this is the only free LHS attribute. The A2A mapping is applied and the evaluation computes no difference between the actual and target models. Thus, the whole mapping process terminates.

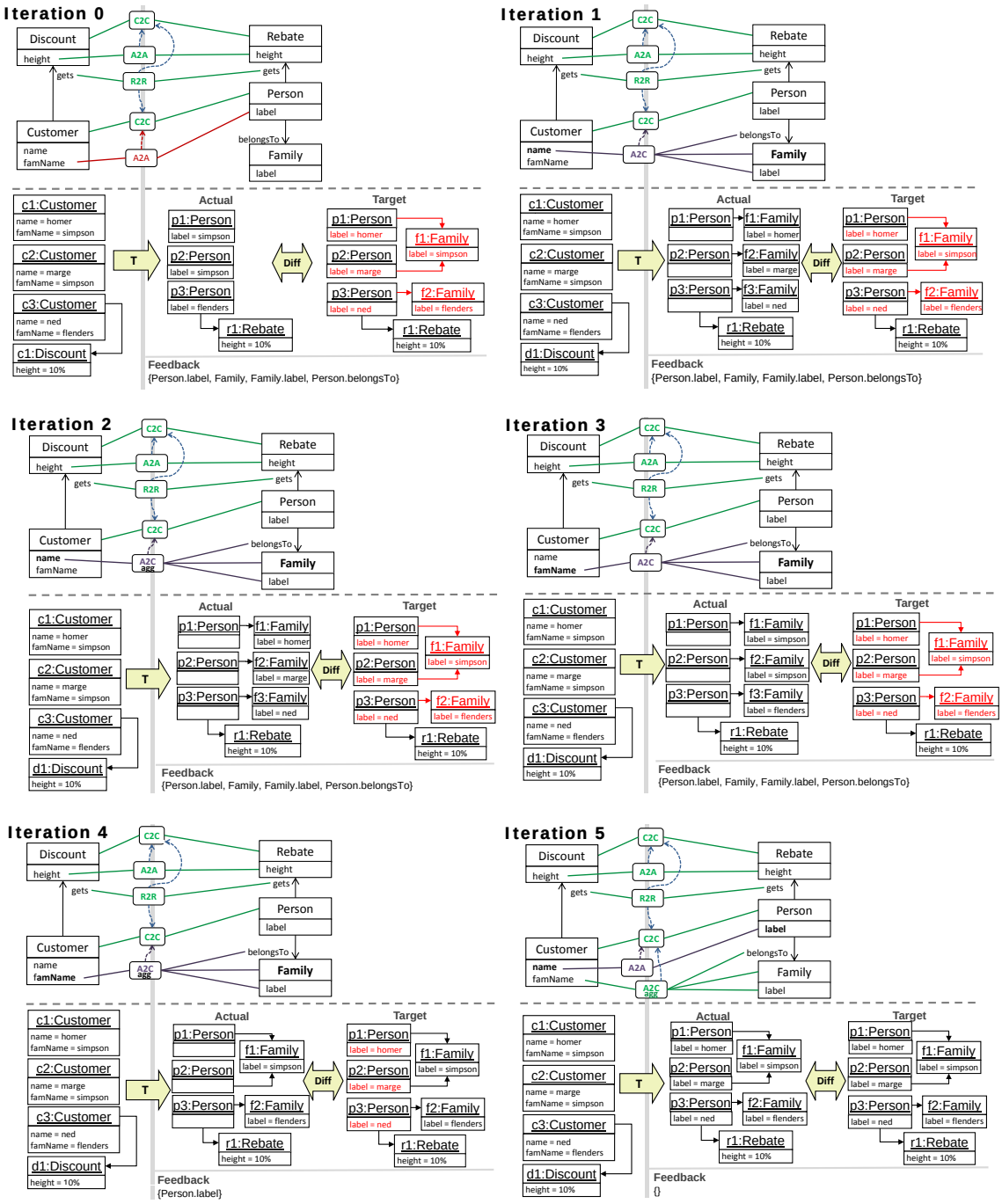

Fig. 4. The Smart Matcher in action.

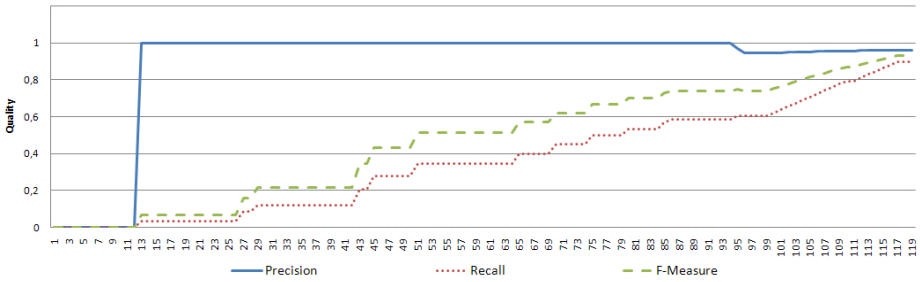

Fig. 5. The development of quality indicators over time.

5 Evaluation

In the project ModelCVS[3] which was aimed at finding solutions for the integration of modeling tools, the need for automatic model transformations arose. Due to the lack of appropriate tools for metamodel matching, and the strong relatedness of metamodels and ontologies [11], models were lifted to ontologies, and ontology matching tools where applied. Unfortunately, the quality of the resulting alignments did not meet our expectations [6] and we could not deduce reliable mappings for model transformation. Consequently, the Smart Matching approach was established. Although developed with a certain application domain in mind, the result was a generic matching tool which is not restricted to metamodels only. In the following we present results of a case study where elements of the UML class diagram 1.4 metamodel are matched to elements of the UML class diagram 2.0 metamodel. Both metamodels consist of about 60 elements. The user-defined instance models describe a gas station with about 40 elements for the source model and about 30 elements for the target model. For a detailed description of the setting please refer to our project page[4].

Starting from an empty alignment set, the Smart Matcher achieves a precision of 0.963 and a recall of 0.897 resulting in an f-measure of 0.929. When we translate the UML class diagram 1.4 terms into German and match it against the English UML class diagram 2.0, we obtain the same results as the Smart Matcher follows an uninterpreted mapping approach. In contrast, COMA++, the most successful system in our matching systems evaluation [6], yields alignments with a precision of 0.833, a recall of 0.583, and an f-measure of 0.683. The results of the German UML class diagram 1.4 matched with the UML class diagram 2.0 clearly indicate the relevance of similarity information for COMA++: the precision decreases to 0.368 and the recall is 0.117 with an f-measure of 0.178. Fig. 5 shows the evolution of precision, recall, and f-measure with respect to the number of Smart Matching iterations. Whereas the precision value tends abruptly towards one, recall and f-measure steadily increase. At iteration 91, the precision slightly decreases. The found mapping is interpreted as correct with respect

[3] www.modelcvs.org
[4] http://big.tuwien.ac.at/projects/smartmatcher

to the provided instance models, although it is not. This is due to our heuristic for references, which is not sufficient when two references which have the same amount of links are between the same classes. For additional information on the case study, we kindly refer again to our project page.

6 Related Work

The enormous demand for matching systems in order to automatically solve information integration problems led to a vast number of different approaches (for surveys cf. [5, 10]). Following the classification of [10] the Smart Matcher realizes a schema-based approach where user-defined instances are employed to verify and to improve the found alignments iteratively. In this sense, the Smart Matcher is based on supervised machine learning techniques. In addition to the source and the target schema, training data consisting of input/output pairs is provided and a relationship model is generated which maps the input instances to the output instances.

Several systems have been proposed which incorporate machine learning techniques in order to exploit previously gathered information on the schema as well as on the data level. For example, LSD, Autoplex, and Automatch use Naive Bayes over data instances, SemInt is based on neural networks, and iMap analyses the description of objects found in both sources by the establishment of a similarity matrix (see [5] for an overview).

The probably closest approach to ours is SPICY. Like the Smart Matcher, SPICY [2] enhances the mapping generation process with a repeated verification phase by comparing instances of the target schema with transformed instances. The similarity in conceptual architecture of these two approaches contrasts to the significant differences in their realization as well as the moment a human user has to intervent. Whereas SPICY only deals with 1:1 and 1:n alignments, the Smart Matcher is able to resolve more complex heterogenities due to the integration of the CAR-language. This results in very distinct search, comparison, and evaluation strategies. Furthermore SPICY assumes that instances of the target schema are a priori available (e.g. from a Web data source). In contrast, for the application of the Smart Matching approach a human user must explicitly engineer the same instances in the source and in the target schema as the Smart Matcher implements the idea of unit testing. When SPICY terminates, it returns a ranked list of transformations which has then to be evaluated and verified by the user. When the Smart Matcher terminates, the resulting mappings are correct and complete with respect to the source and target input instances.

Finally, the eTuner [9] approach automatically tunes the configurations of arbitrary matching systems by creating synthetic schemas for which mappings are known. Hence, in contrast to the Smart Matcher which tunes the search process itself, the eTuner adjusts other systems externally in order to increase the accuracy of the found alignments.

7 Conclusion and Future Work

In this paper we have presented the realization of the Smart Matcher, which implements a generic method for the automatic detection of executable mappings between a source schema and a target schema. During the whole search process, human intervention is only necessary once, namely at the beginning. The user has to define instances which represent the same real world issue in the language of the source schema as well as in the language of the target schema. Those instances allow a permanent evaluation and an incremental improvement of the found mappings by the comparison of the user given target instances with the generated actual instances. The Smart Matcher's feedback-driven approach guarantees that the resulting mappings are sound and complete with respect to those instances. Therefore an accurate engineering of the instances is inevitable for the successful application of the Smart Matcher, but the effort spent in the preparation phase is compensated with less effort necessary in later phases. For many schemas (e.g., UML diagrams, ontologies, database schemas) the creation of the instances is supported by comfortable editors and once the instances are established for one schema, those instances may be reused in multiple integration scenarios. Furthermore, the required size of the instances does not need to be large for obtaining a satisfying quality of the mappings, as we have experienced in our case study.

In future work, we will focus on the development of appropriate methods for the definition of the example instances in order to provide recommendations and in order to guide the user during the preparation phase. Furthermore, we plan more extensive comparisons with other systems, especially with those which incorporate learning techniques. Concerning the Smart Matcher itself, we plan to improve the search strategies by enhancing them with sophisticated heuristics.

References

1. P. A. Bernstein and S. Melnik. Model Management 2.0: Manipulating Richer Mappings. In *Proc. of the 2007 ACM SIGMOD Int. Conf. on Management of Data*, pages 1–12. ACM Press, 2007.
2. A. Bonifati, G. Mecca, A. Pappalardo, S. Raunich, and G. Summa. Schema Mapping Verification: The Spicy Way. In *Proc. of the 11th Int. Conf. on Extending Database Technology (EDBT'08)*, pages 85–96. ACM, 2008.
3. H. Erdogmus and T. Morisio. On the Effectiveness of Test-first Approach to Programming. *IEEE Transactions on Software Engineering*, 31(1):226–237, 2005.
4. J. Euzenat. An API for Ontology Alignment. In *Proc. of the 3rd Int. Semantic Web Conference (ISWC'04)*, pages 698–712. Springer LNCS, 3298, 2004.
5. J. Euzenat and P. Shvaiko. *Ontology Matching*. Springer, 2007.
6. G. Kappel, H. Kargl, G. Kramler, A. Schauerhuber, M. Seidl, M. Strommer, and M. Wimmer. Matching Metamodels with Semantic Systems—An Experience Report. In *Workshop Proc. of the 12th GI-Fachtagung Datenbanksysteme für Business, Technologie und Web (BTW'07)*, pages 38–52. Verlag Mainz, 2007.

7. G. Kappel, H. Kargl, T. Reiter, W. Retschitzegger, W. Schwinger, M. Strommer, and M. Wimmer. A Framework for Building Mapping Operators Resolving Structural Heterogeneities. In *Proc. of the 2nd Int. United Information Systems Conf. (UNISCON'08)*, pages 158–174. Springer LNBIP, 5, 2008.
8. H. Kargl and M. Wimmer. SmartMatcher—How Examples and a Dedicated Mapping Language can Improve the Quality of Automatic Matching Approaches. In *Proc. of the 2nd Int. Conf. on Complex, Intelligent and Software Intensive Systems (CISIS'08)*, pages 879–885. IEEE Computer Scociety, 2008.
9. Y. Lee, M. Sayyadian, A. Doan, and A. Rosenthal. eTuner: Tuning Schema Matching Software Using Synthetic Scenarios. *VLDB Journal*, 16(1):97–122, 2007.
10. E. Rahm and P. Bernstein. A Survey of Approaches to Automatic Schema Matching. *VLDB Journal*, 10(4):334–350, 2001.
11. M. Wimmer, T. Reiter, H. Kargl, G. Kramler, E. Kapsammer, W. Retschitzegger, W. Schwinger, and G. Kappel. Lifting Metamodels to Ontologies—a Step to the Semantic Integration of Modeling Languages. In *Proc. of the 9th Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS'06)*, pages 528–542. Springer LNCS, 4199, 2006.