Normalizing Heterogeneous Service Description Models with Generated QVT Transformations*

Simon Schwichtenberg, Christian Gerth, Zille Huma, and Gregor Engels

s-lab - Software Quality Lab, University of Paderborn, Germany
{simon.schwichtenberg,gerth,zille.huma,engels}@upb.de

Abstract. Service-Oriented Architectures (SOAs) enable the reuse and substitution of software services to develop highly flexible software systems. To benefit from the growing plethora of available services, sophisticated service discovery approaches are needed that bring service requests and offers together. Such approaches rely on rich service descriptions, which specify also the behavior of provided/requested services, e.g., by pre- and postconditions of operations. As a base for the specification a data schema is used, which specifies the used data types and their relations. However, data schemas are typically heterogeneous wrt. their structure and terminology, since they are created individually in their diverse application contexts. As a consequence the behavioral models that are typed over the heterogeneous data schemas, cannot be compared directly. In this paper, we present an holistic approach to normalize rich service description models to enable behavior-aware service discovery. The approach consists of a matching algorithm that helps to resolve structural and terminological heterogeneity in data schemas by exploiting domain-specific background ontologies. The resulting data schema mappings are represented in terms of Query View Transformation (QVT) relations that even reflect complex n:m correspondences. By executing the transformation, behavioral models are automatically normalized, which is a prerequisite for a behavior-aware operation matching.

Keywords: SOA, Service Description, Ontologies, Behavioral Models, Matching.

1 Introduction

Due to their modularity, reusability, and flexibility, SOAs allow to realize software projects faster and may reduce development costs drastically. In such service-oriented scenarios, service providers offer services and service requesters request for services. The process to match service offers (SOs) with service requests (SRs) is called service discovery.

A service discovery that is performed manually is time-consuming and error-prone, since a user has to understand and compare all SOs and SRs individually. In addition, misinterpretations and misunderstandings of the services lead to

* This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901).

J. Cabot and J. Rubin (Eds.): ECMFA 2014, LNCS 8569, pp. 180–195, 2014.
© Springer International Publishing Switzerland 2014

inaccurate matching results. For this reason and to benefit from the plethora of existing services, an automatic service discovery is required that is based on comprehensive specifications of the services.

Existing service specification languages like Web Ontology Language for Web Services [15], Web Service Modeling Language [5], Semantic Annotations for WSDL and XML Schema [18], and the Rich Service Description Language [10,9] allow to create such comprehensive specifications. Typically, such a service specification includes a structural data schema and behavioral models, e.g., in form of visual contracts (VCs) [6]. VCs describe the behavior of SRs or SOs in terms of pre- and postconditions for their respective operations. The data schema specifies the used data types of the service and their relations. The behavioral models in turn are typed over the data schema. For the remainder, we assume that such a data schema is specified in terms of a Unified Modeling Language (UML) class model and describes the relevant concepts of a certain domain, e.g., tourism or banking. Consequently, complex data types are referred to as classes, primitive types as attributes, and instances of types as objects.

Since SOs and SRs are created independently, the structure and terminology of their class models are most likely heterogeneous, even if they specify services in the same domain of interest. As a consequence, the behavioral models typed over the class models cannot be compared directly and a behavior-aware operation matching of service requests and offers is not possible. The heterogeneity of the class models arises from different terminologies, granularity levels, and logical structuring. For example, two classes of the SR's and the SO's class models might have different but synonymous identifiers, since both denote the same concept. Analogously, homonyms must be addressed separately.

Matching classes and attributes is an important aspect to overcome the heterogeneity, e.g., to determine whether parameters and return values of provided and requested operations correspond to each other. Thereby, a single class does not necessarily have to correspond to a single other class. It is rather likely that sets of classes correspond to each other, resulting in complex 1:n, n:1, or even n:m mappings. However, complex mappings have received little attention in ontology matching approaches [19,16].

In this paper, we present an holistic approach to resolve the structural and terminological heterogeneity of rich service description models to enable a behavior-aware service discovery and composition. Our approach includes a class model matching algorithm that leverages domain-specific background ontologies, e.g., linguistic resources and the Semantic Web to establish semantic relations between similar classes across different models. From the obtained class model mappings, a QVT [1] script is generated automatically, whose particular relations reflect identified correspondences between classes of any cardinality. Using these QVT relations, we normalize behavioral models in rich service descriptions and make them directly comparable in order to enable a behavior-aware operation matching.

The remainder is structured as follows: In Sect. 2, we describe a scenario that illustrates the problem statement. Our approach for class model matching and

VC normalization is presented in Sect. 3 and Sect. 4. Tool support is introduced in Sect. 5. We discuss related work regarding class model and web service matching in Sect. 6. Finally, in Sect. 7 we conclude and give an outlook on future work.

2 Scenario

For the following, we assume that a service requester wants to create a service that allows its users to search and book hotel rooms. Therefore, the requester is interested in SOs of hotel chains that provide access to their room availability data or functionality to make bookings through a service interface. For that purpose, the requester specifies a SR, which is then compared with available SOs of hotel chains. We assume that SRs and SOs are specified using a service specification language, which consists at least of the following parts: (1) a UML class model specifying the data schema and (2) visual contracts, which specify pre- and postconditions for every requested and provided operation.

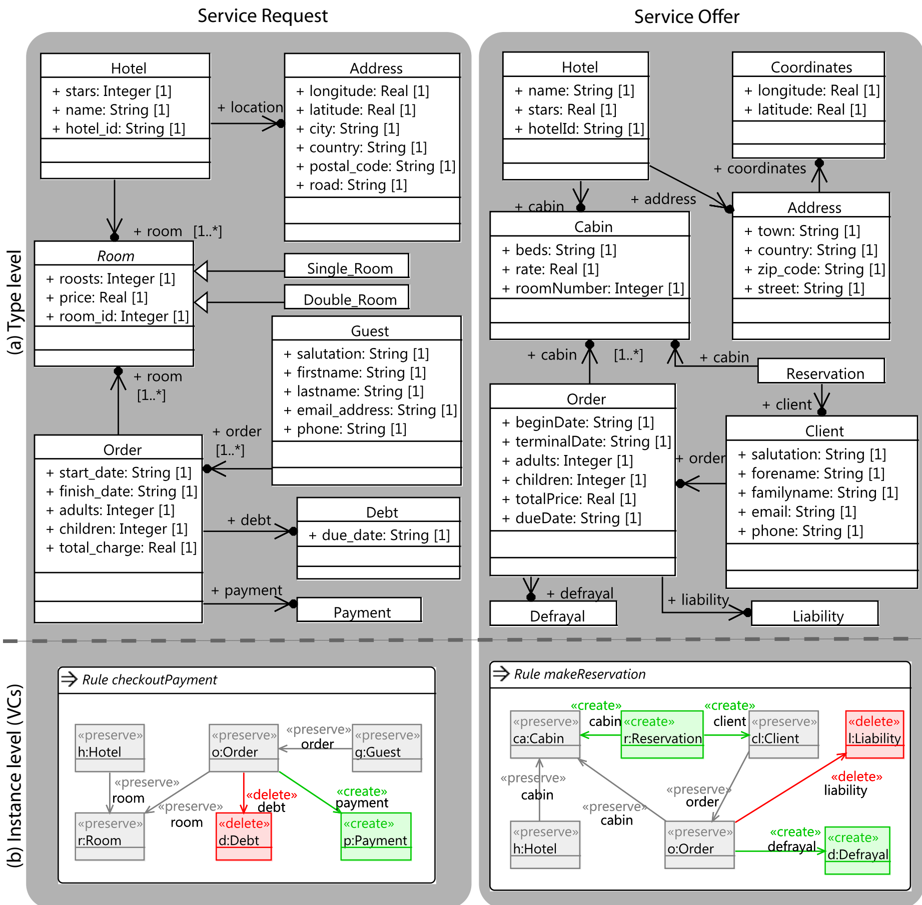

Fig. 1. Heterogeneous Class Models and Visual Contracts

Fig. 1 gives an example of such a SR and SO with their respective class models and two behavioral models in terms of visual contracts.

Obviously, both class models have some concepts in common. However, both use partially different terminologies for these concepts, e.g., the classes `Debt` and `Liability`. Further, classes do not necessarily correspond only to a single other class. For instance, `Single_Room` and `Double_Room` are specializations of `Room` in the SR, while there are no further specializations of `Cabin` in the SO. In addition, some attributes are widespread differently over several classes in both class models, resulting in complex 1:n, n:1, and n:m correspondences between classes. For instance, the attributes of `Address` in the SR are represented by `Address` and `Coordinates` in the SO.

The behavioral aspects of a service are described using VCs. A VC is a graph grammar rule, whose left-hand side (LHS) describes a precondition that must be fulfilled before a certain operation can be executed. The right-hand side (RHS) of the rule describes the effects of the operation execution. The graphs of the rule's LHS and RHS are instances of the SR or SO class model.

Fig. 1 (b) shows the VCs of the requested operation `checkoutPayment` and the provided operation `makeReservation`. The VC of `checkoutPayment` specifies the following behavior:

> After `Guest` g has paid the `Debt` d for his `Order` o of `Hotel` h's `Room` r, `Debt` d is deleted and a new `Payment` p is created instead.

This operation is similar to `makeReservation` that uses synonymous identifiers and additionally creates a `Reservation` after the payment. However, in order to match these models to decide whether the behavior of the operations is equivalent, the behavioral models need to be normalized. For that purpose, we propose the approach shown in Fig. 2.

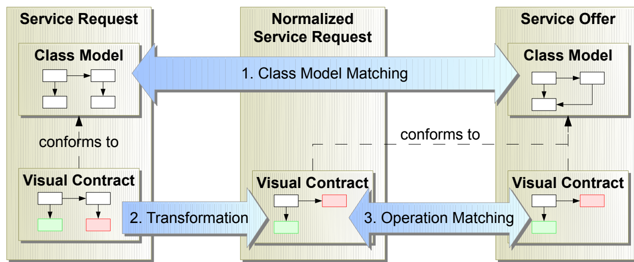

Fig. 2. Normalization Approach

Our approach consists of three steps: (1) First, we determine class and attribute mappings during *Class Model Matching* by considering background ontologies in order to overcome structural and terminological heterogeneity. (2) From the set of mappings, we generate a relational QVT script. By executing the `Transformation`, all VCs of the requester are retyped according to the class model of the SO. (3) After the normalization, the operations can be matched, e.g., by using the approach introduced in [10].

After the computation of a mapping between the class models of a SR and SO, we assume that the result is inspected by a user, since it may happen that manual intervention is required. For instance, in case of attributes with the same name but different types it is necessary to parse their values properly, e.g., by converting a float to an integer or by integrating an adapter. In such cases, the QVT script must be adjusted manually. To give an example: The registry might return a list of the ten best matching offers to the requester. The requester selects one of them and refines the QVT script if needed.

In the following sections, we describe the class model matching and the normalization in detail.

3 Class Model Matching

The following section introduces our approach for an automatic service discovery (SD). At first, the architecture and the process of the SD are defined. The rest of this section focuses on the class model matching and exemplifies the matching algorithm.

3.1 Automatic Service Discovery Process

In our approach, there are three parties that take part in the SD: service providers, service requesters, and a service registry. The registry is a central point to convey SRs to SOs. Providers publish specifications of their SOs at the service registry. Requesters inquiry the registry by SRs that are also specified as described in the previous section. The registry resolves the class model heterogeneity on-the-fly, retrieves matching SOs for the SR, and returns them to the requester. The requester can bind its SR to one of the proper SOs. The SD process is described in the following section.

Fig. 3 illustrates the matching process that is executed at the registry. The single process steps are described in the following.

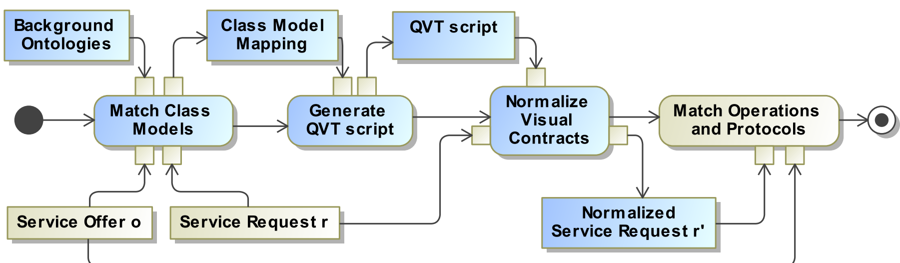

Fig. 3. Automatized Matching Process

Match Class Models: In the first step, the class models of all registered SOs and the currently inquired SR are matched. The class models are enriched with additional information from several background ontologies (BOs) to establish mappings between classes and attributes. The output of this step is a set of disjoint mappings that may have any kind of cardinality. Complex mappings enable a relation between n classes of the SO's class model and m classes of the SR's class model, with $n, m \leq 1$.

Generate QVT script: In the next step, a relational QVT script is generated from the set of mappings. Each QVT relation corresponds to a class mapping. If the classes of a particular mapping have attributes that were also mapped for their part, these attribute mappings are likewise considered in these relations. However, class and attribute mappings are considered in isolation, i.e. the mapped attributes do not necessarily have to be contained in classes of the same class mapping.

There are still some aspects of heterogeneity our class model matcher does not cover yet. In fact, more complex than 1:1 attribute mappings, association mappings, and correspondences between attributes and classes are not considered. Nevertheless, our class model matcher helps to resolve heterogeneity across class models, but we assume that the identified mappings are inspected by a user and adjusted if necessary before the script is executed to normalize visual contracts.

Normalize Visual Contracts: To normalize the requester's VCs, the QVT script is executed with these VCs as input. After the transformation, the SR's VCs conform to the class model of the SO. The normalization transforms the LHS and RHS of each VC separately and composes the results to a single normalized VC. It should be noticed, that it is also possible to normalize the VCs according to the class model of the SO, because relational QVT allows bidirectional transformations. As a useful by-product, the transformation can be used for a mediation service: Parameters of the SR as well as return values of the SO can be translated on-the-fly, when the SR and SO interact.

Match Operations and Protocols: Until now, only structural aspects of the SD have been addressed. On the contrary, operation matching takes the behavioral models of SRs and SOs into account. Our approach integrates the operation matching approach proposed by Huma et al. [10], which relies on VCs. The approach considers n-ary correspondences between operations. For example, a 1:1 operation mapping is established when the precondition of the requested operation covers at least the preserved and deleted objects of the provided operation and the postcondition of the provided operation covers at least the preserved and created objects of the requested operation. More complex matching strategies consider whole sequences of operations.

The approach of Huma et al. also comes with a protocol matching, which is out of scope in this paper. The following section concentrates on the service discovery step *Match Class Models*. The algorithm and its interplay with background ontologies is explained in detail.

3.2 Matching Algorithm

The following section exemplifies the class model matching algorithm which is a prerequisite for the normalization of the VCs. Fig. 4 shows the single steps of the algorithm that are explained in detail in the remainder.

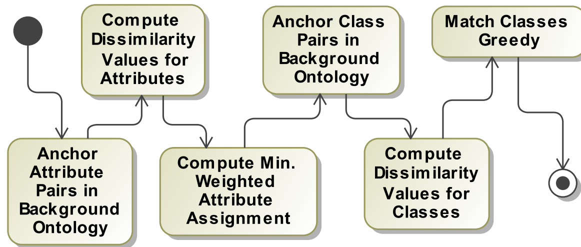

Fig. 4. Class Model Matching Algorithm

Anchor Attribute Pairs in Background Ontology: Class models of service descriptions are compact, because they focus on the implementation, abstract from irrelevant details, and ideally do not contain redundant information. Consequently, the information class models contain is typically not sufficient to establish semantic relations across different class models. Additional knowledge is required to establish these relations. In our approach, the class models are embedded in BOs (c.f. [3]), which model particular domains of interest more holistically than class models and hopefully builds bridges between the different class models.

Our approach is not limited to a certain ontology language or to a certain domain of interest. Rather any kind of ontologies can be used as long as they classify their concepts in a taxonomy. In particular, the algorithm can access common knowledge ontologies like the linguistic resource WordNet [14], the DBpedia ontology [13], and further Web Ontology Language (OWL) ontologies like Schema.org[1] or Umbel[2]. Furthermore, we included some domain-specific OWL ontologies with regard to the previously described scenario, e.g. OnTour[3] and the Travel Guide[4]. In order to support a certain ontology language, a respective programming interface must be implemented that returns the hypernyms for a given term. In addition, the ontologies can be imported into a database that maps terms to their hypernyms, which avoids to keep the whole ontology in memory and accelerates the anchoring step by leveraging database indexes.

During the anchoring, a BO is selected that contains two concepts with the same identifiers as the two attributes to be matched. This means a BO is selected for each individual attribute pair. The identifiers of the concepts and the attributes need not to be exactly the same: Different naming conventions like

[1] http://schema.org/
[2] http://umbel.org/
[3] http://e-tourism.deri.at/ont/
[4] https://sites.google.com/site/ontotravelguides

`due_date` and `dueDate` are also considered. For this purpose we use tokenization and elimation of stopwords like *of, the, a*, etc. [7]. Fig. 5 shows an anchoring of the attributes `price` and `rate` within the taxonomy of WordNet. By using different BOs for matching, it may happen that homonyms are misinterpreted if their semantics differs between the BOs. Although, in practice the number of homonyms shared between different domains of interest is minimal, homonyms may be inspected manually after the matching process.

Fig. 5. Anchoring of `rate` and `price` in WordNet

Compute Dissimilarity Values for Attributes: It is a common technique to express the dissimilarity of two concepts as a number. The lower the value of the number, the less dissimilar the concepts are. The normalized dissimilarity is a function from a pair of model elements to a number that ranges over the unit interval of real numbers (c.f. [7]).

In the current step, all dissimilarity values for attributes are determined pairwise. Once two attributes have been anchored, their degree of dissimilarity is determined by the upward cotopic dissimilarity [12], which is defined in Def. 1 (c.f. [7]).

Definition 1. ***Upward cotopic dissimilarity.*** *The upward cotopic dissimilarity* $\delta : o \times o \rightarrow [0, 1]$ *is a dissimilarity over a hierarchy* $H = \langle o, \leq \rangle$*, such that:*

$$\delta(c, c') = 1 - \frac{|UC(c, H) \cap UC(c', H)|}{|UC(c, H) \cup UC(c', H)|}$$

where $UC(c, H) = \{c' \mid \forall c, c' \in H \wedge c \leq c'\}$ *and* $c \leq c'$ *means that* c' *is more general than* c*.*

The upward cotopic dissimilarity relates the number of shared hypernyms to the total number of hypernyms of two concepts to be matched according to the BO. The shared hypernyms of `price` and `rate` according to WordNet are highlighted in Fig. 5, which yields a dissimilarity value of $\delta = 1 - 5/10 = 0.5$.

The primitive type compatibility of the attributes is also taken into account when their dissimilarity is assessed. Accordingly, two attributes with numerical types are less dissimilar than a numerical and an alphabetical type. The dissimilarity of the types is encoded in a static look-up table. The upward cotopic dissimilarity and the type compatibility are aggregated into a single dissimilarity value according to Equation 1, which aggregates N different dissimilarity values δ_i, where each of them is weighted by ω_i. Fig. 6 shows the dissimilarity values of `city` to all the attributes of the other class model.

$$\hat{\delta} = \sum_{i=1}^{N} \delta_i \omega_i, \quad \sum_{i=1}^{N} \omega_i = 1, \quad \forall \omega_i : \omega_i > 0 \tag{1}$$

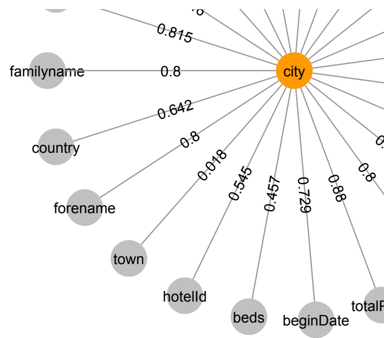

Fig. 6. Dissimilarity values for `city`

Compute Min. Weighted Attribute Assignment: After the dissimilarity values have been computed for all attribute pairs, proper attribute mappings are created. We assume that most of the attributes represent atomic information, which is why only 1:1 attribute mappings are considered in our approach. The mapping of attributes is considered as an optimization problem with the aim to assign as many attributes as possible while minimizing the sum of the dissimilarity values. The matching algorithm uses an existing algorithm for the minimum cost flow problem [20] as a subroutine to find this minimal weighted assignment. A solution for the assignment referring to the scenario is shown in Fig. 7.

Fig. 7. Min. Weighted Attribute Assignment

Anchor Class Pairs in Background Ontology: This step is analogous to the anchoring of attributes.

Compute Dissimilarity Values for Classes: The dissimilarity of the class pairs is assessed by using the upward cotopic dissimilarity. In addition, the attribute mappings are also taken into account. The intuition is that classes that share many

Hotel
Payment
Debt
Address
Order

Fig. 8. Shared Attribute Dissimilarity

Fig. 9. Class Mappings

dissimilar attributes are also dissimilar for their part. The shared attributes dissimilarity relates the number of shared attributes to the maximal number of attributes. Fig. 8 shows the attribute mappings between `Room` and `Cabin` that are part of the optimal attribute assignment. Thus, the shared attribute dissimilarity is $\delta = \frac{0.6+0.24+0.788+3}{6} \approx 0.771$. The upward cotopic and the shared attributes dissimilarity are aggregated into a single dissimilarity value in accordance with Equation 1.

Match Classes Greedy: Contrary to the attribute matching, the classes are matched with a greedy strategy instead of calculating an optimal assignment. The class mappings are created in a ascending order according to the dissimilarity of the class pairs. At the same time, mappings are only created as long as the dissimilarity is below a threshold. If one of a pair's classes is already part of a mapping, the other is added to that mapping. That way 1:1 mappings can be expanded to complex 1:n, n:1, or n:m mappings. Fig. 9 shows the resulting set of complex class mappings. Consequently, the set of resulting class mappings is disjoint.

The set of attribute and class mappings that link the SR's and the SO's class models is the input for the VC normalization, which is the subject of the next section.

4 Visual Contract Normalization

As a preparation for the normalization of the VCs, a QVT [1] script is generated from the mappings obtained by the class model matching. Because of its relational character, the QVT script can be executed in both directions. Thereby the transformation direction is determined implicitly by selecting the target model. In our approach, we transform the VCs that conform to the class model of the SR into VCs that conform to the SO's class model. The LHS and the RHS of a VC are transformed separately and recomposed into a newly created, normalized VC. This section describe how the transformation script is generated. The generation process consists of three steps: Relations are (1) created for the class mappings, (2) enriched by attribute mappings, (3) connected with respect to associations.

Class Mappings: Fig. 10 shows the QVT relation that was generated from the 1:2 class mapping between the classes `Reservation` and `Order` of the SO's class model and the class `Order` of the SR's class model. Each class mapping corresponds to a **top relation**. The **top** keyword indicates that the relation is an entry point of a transformation, which means that object bindings from other relations are not required. For each of the classes of the mapping, a corresponding **domain** is added to the relation. A relation domain has a *domain pattern* that describes a specific model graph consisting of objects, their attributes, and association links. A relation holds, when all its domain pattern match in the source and target model respectively. In case a relation holds, the *root variables* of the domains are bound to concrete objects. The **enforce** keyword ensures that a relation holds when the domains of the source model can be bound. If necessary, new objects of the target model are created or existing objects are deleted. As a result, the source and target model are consistent in regard to that relation.

```
                debt = var_debt : requester::Debt {},
                total_charge = var_totalPrice ...
15          };
            when {
                var_dueDate = if dom_order1.oclIsUndefined()
                then dom_order2.debt.due_date
                else dom_order1.dueDate
20              endif;
                Client_Guest(var_client, requester::Guest.allInst
            }
        }
        top relation Address_Coordinates_Address{
25          enforce domain provider dom_address1 : provider::Addres:
                coordinates = dom_coordinates
            };
            enforce domain provider dom_coordinates : provider::Coo
            enforce domain requester dom_address2 : requester::Addr
30      }
```

Fig. 10. Excerpt of the Generated QVT Script

Attribute Mappings: After the generation of relations, we add appropriate QVT statements to the relations that take the attribute mappings into account. Each 1:1 attribute mapping corresponds to a *Property Template Item (PTI)* that is added to the respective domain pattern. In our approach, we use PTIs to instruct the transformation where to read attributes values from the source model and where to assign these values to attributes of the target model. Since classes and attributes were matched independently, it may happen that it is not possible to reflect class and attribute mappings at the same time. We identified two cases where we added PTIs to relations: (1) Both mapped attributes are owned by two classes that were mapped for their part. As shown in Fig. 10, the classes `Order`

were mapped and are part of the same relation. Their attributes `totalPrice` and `total_charge` were also mapped. Hence, respective PTIs are added that bind these attributes to the same value over variable `var_totalPrice` (line 9 and 14). (2) Both mapped attributes are owned by two classes that have not been mapped for their part. For example, the attributes `dueDate` and `due_date` were mapped. Their owning classes `Order` and `Debt` have not been mapped for their part. However, `due_date` can be accessed over `dom_order2` because `Order` has an association to `Debt`. Hence, a PTI is added to the respective domain pattern that binds the value of `due_date` to the variable `var_dueDate` (line 10). Furthermore, `var_dueDate` is bound to an Object Constraint Language (OCL) [2] expression in the **when** clause of the relation. The expression represents the navigation path that accesses `due_date` (line 18). It should be noticed, that the navigation path depends on the transformation direction and that such a path may only exist in one direction. The transformation direction can be determined by calling the OCL function call *dom_order1.oclIsUndefined()*. Thus, `var_dueDate` is only bound to `due_date` **if** transforming from the SR's to the SO's class model. The **else** branch has no effect and is just for syntactical validity.

Associations: Until now, we considered classes in isolation during the QVT script generation. Here we examine associations between classes, i.e. association instances from the source model are transformed to links of the target model. During the class model matching (Sect. 3) no association mappings were established that could be translated to QVT statements. Instead, the link creation is exclusively derived from the class mappings. To simplify, we assume that classes have at most one unidirectional, single-valued association to another certain class. Links between objects are established by either referencing already bound variables that are available in the scope of a relation or by binding variables with relation calls. Similar to the attribute mappings, we distinguish two cases when and how to create links: (1) A complex class mapping maps n classes of a class models to m other classes. One of the n has an association to one of the other $n-1$ classes. For example, `Address` has an association to `Coordinates` and both classes are part of the same mapping. Hence, a PTI is added to the respective `Address` domain pattern, that binds the association `coordinates` and the root variable `dom_coordinates` (line 26). (2) A class has an association to another class and both classes were not mapped. For example, `Reservation` has an association to `Client` and both classes were not mapped. Hence, a PTI is added to the `Reservation` domain pattern that binds the association `client` to an *object template expression* which describes the characteristics of a specific `Client` object (line 5). The object template is empty, because its characteristics are already defined in the `Client` domain pattern, that occurs in another relation. Thus, the binding of the variable is delegated to the other relation by a *relation call expression* in the **when** statement (line 21). The parameter types of a *relation call* must conform to the types of the domains of the called relation. If conforming variables are available in the scope of the calling relation, they are also used as parameters in relation calls. Otherwise, any arbitrary instance of a required

parameter type is used as a parameter. Exemplary, such arbitrary objects are determined by calling the OCL function *Guest.allInstances()->any(true)*.

To summarize, we have shown how a QVT transformation script can be generated automatically from the set of mappings. The next section introduces our implementation for the class model matcher and the QVT script generator.

5 Tool Support

We integrated our service description model normalization into the *Rich Service Description Language (RSDL) Workbench*, which allows to create, publish, and discover Rich Service Description Language (RSDL) specifications. The RSDL workbench is realized as a plug-in for the integrated development environment Eclipse and is under development at the University of Paderborn in the Collaborative Research Centre 901 *On-the-Fly Computing*. Our workbench uses the following third-party plug-ins: Papyrus[5] that provides a graphical editor for the UML parts of the specifications and Henshin[6] that provides a graphical editor that is used to model the VCs. The service discovery was realized as a Java API for RESTful Web Services (JAX-RS) by means of the Jersey framework[7].

Further, our class model matcher uses the Eclipse plug-in EMF Compare[8]. The indexing mechanism is based on the Jena framework[9] to read background ontologies, which are stored in a normalized form in a database. For the QVT script generation, the metamodel of the QVTd[10] project has been used. The transformation execution engine mediniQVT[11] is leveraged to run the transformation in order to normalize the VCs. Fig. 11 shows the identified mappings between two class models of a service request and a service offer in the graphical user interface provided by EMF Compare.

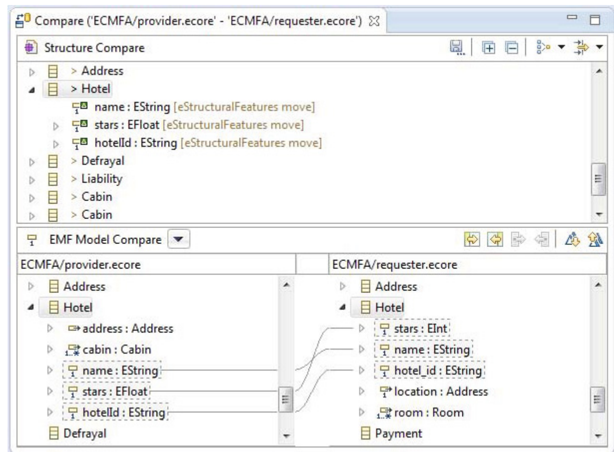

Fig. 11. Graphical User Interface (EMF Compare)

[5] http://www.eclipse.org/papyrus/
[6] http://www.eclipse.org/henshin/
[7] https://jersey.java.net/
[8] http://www.eclipse.org/emf/compare/
[9] http://jena.apache.org/
[10] http://www.eclipse.org/mmt/qvtd
[11] http://projects.ikv.de/qvt

6 Related Work

Several research areas are related to the normalization of service description models: ontology and model matching, coupled model evolution, and service discovery. Concerning ontology matching, many approaches and systems have been (and still are) developed that allow to create mappings between heterogeneous ontologies or schemas. A nice overview of various approaches and the current state of research is provided by the survey [19].

A classification of matching approaches is provided by [11], which distinguishes between: static identity-based, signature-based, similarity-based, and language-specific matching approaches. According to this classification, our class model matching approach is a similarity-based matching approach, which additionally leverages background ontologies to identify synonyms, homonyms, as well as correspondences between classes with a similar ontological semantic.

In the following, we want to highlight three matching systems exemplary: EMF Compare identifies differences between different versions of the same Eclipse Modeling Framework (EMF) model and is therefore suited for an integration in version control systems to keep track of the model's evolution. The matching algorithm behind EMF Compare is related to [21]. However, without adaptations EMF Compare is not suited to match heterogeneous class models, because it does not recognize synonyms for example. The matching process of Scarlet [17] anchors the concepts to be matched in one or more background ontologies that were determined by external Semantic Web search engines. Next, a semantic relation between the matched concepts is inferred from the anchor concepts. However, similarity values are not computed. Furthermore, most of the matching systems consider only 1:1 mappings. An exception is e.g., AgreementMaker [4], which first computes all similarity values and afterwards a user selects the desired metrics and mapping cardinalities. Then, an algorithm is executed to compute an optimal solution on the weighted assignment problem. This algorithm is iteratively executed to obtain n:m mappings.

In the case of class models, Coupled Model Evolution (CME) addresses the problem, that object models become inconsistent when their class model evolves. In this regard, CME is related to the VCs normalization except that CME considers different versions of the class model, whereas VCs normalization considers heterogeneous class models. COPE [8] is an approach that keeps track of class model modifications and creates a history of changes. A migrator that is generated from this history applies the changes to the object models. COPE is not suited to normalize VCs, because typically no such change history is available for heterogeneous class models.

The service discovery approach that was introduced in this thesis relies on the approach of Huma et al. [10] that uses RSDL specifications [9]. Huma et al. also address heterogeneous data schemas: As a preparation for an automation of the heterogeneity resolution, the approach requires that local class models are manually mapped to a common global class model. Our approach aims to close this gap and automate this process as far as possible.

7 Conclusion and Future Work

In this paper, we introduced an holistic approach to overcome structural as well as terminological heterogeneity of service description models. Thereby, we enable the behavior-aware matching of service requests and services offers.

The main contributions of our work are as follows: In contrast to yet another structural model matcher, our class model matcher exploits domain-specific background ontologies that offer the opportunity to identify semantic relations between different class models. While the majority of existing matchers only consider 1:1 class correspondences, our matcher identifies also complex 1:n, n:1, and n:m correspondences that arise from different logical structuring or from different degrees of granularity. By representing identified class mappings in terms of QVT relations, we enable an automatic normalization of the behavioral model, which are typed over the class models.

In future work, we will conduct an extensive evaluation of our class model matcher in the course of the CRC 901 and by participating in the Ontology Alignment Evaluation Initiative (OAEI)[12] that aims for a systematic evaluation of matching systems.

Furthermore, we intent to address current limitations of our approach. Concerning the class model matcher these are the identification of mappings between attributes and classes as well as associations mappings. In the former case, an attribute in one class model does not necessarily correspond to another attribute, because the same information an attribute represents may be derived from a class. An example of the latter case is represented by a class that has more than one association to another class, where each represents a different role. In such situations association mappings are necessary.

Concerning the transformation script generation, multi-valued associations are not considered yet. The transformation allows to establish links to at most one object, whatever cardinality the respective association has. Furthermore, the class model matcher allows in fact to map attributes with different types. However, manual intervention is required to reasonably translate, e.g., an alphabetic to a numeric value. Finally, complex QVT relations appeared as a problem, because the more domains a relation has, the less likely it is that the domain patterns will match and the relation will hold. Further research is required to determine a proper number of maximal domains.

References

1. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.1 (January 2011), `http://www.omg.org/spec/QVT/1.1/PDF/`
2. OMG Object Constraint Language (OCL) Version 2.3.1 (January 2012), `http://www.omg.org/spec/OCL/2.3.1/PDF`

[12] `http://oaei.ontologymatching.org/`

3. Aleksovski, Z., Klein, M., ten Kate, W., van Harmelen, F.: Matching Unstructured Vocabularies using a Background Ontology. In: Staab, S., Svátek, V. (eds.) EKAW 2006. LNCS (LNAI), vol. 4248, pp. 182–197. Springer, Heidelberg (2006)
4. Cruz, I.F., Antonelli, F.P., Stroe, C.: AgreementMaker: Efficient Matching for Large Real-World Schemas and Ontologies. Proc. of the VLDB Endowment 2(2), 1586–1589 (2009)
5. de Bruijn, J., Lausen, H., Polleres, A., Fensel, D.: The Web Service Modeling Language WSML: An Overview. In: Sure, Y., Domingue, J. (eds.) ESWC 2006. LNCS, vol. 4011, pp. 590–604. Springer, Heidelberg (2006)
6. Engels, G., Güldali, B., Soltenborn, C., Wehrheim, H.: Assuring Consistency of Business Process Models and Web Services Using Visual Contracts. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE 2007. LNCS, vol. 5088, pp. 17–31. Springer, Heidelberg (2008)
7. Euzenat, J., Shvaiko, P.: Ontology Matching, vol. 18. Springer, Heidelberg (2007)
8. Herrmannsdoerfer, M., Benz, S., Juergens, E.: COPE - Automating Coupled Evolution of Metamodels and Models. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 52–76. Springer, Heidelberg (2009)
9. Huma, Z., Gerth, C., Engels, G., Juwig, O.: A UML-based Rich Service Description Language for Automatic Service Discovery of Heterogeneous Service Partners. In: CAiSE Forum, pp. 90–97 (2012)
10. Huma, Z., Gerth, C., Engels, G., Juwig, O.: Towards an Automatic Service Discovery for UML-based Rich Service Descriptions. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MODELS 2012. LNCS, vol. 7590, pp. 709–725. Springer, Heidelberg (2012)
11. Kolovos, D.S., Ruscio, D.D., Pierantonio, A., Paige, R.F.: Different Models for Model Matching: An analysis of approaches to support model differencing. In: Proceedings of CVSM 2009 @ ICSE 2009, pp. 1–6. IEEE Computer Society (2009)
12. Maedche, A., Zacharias, V.: Clustering Ontology-based Metadata in the Semantic Web. In: Elomaa, T., Mannila, H., Toivonen, H. (eds.) PKDD 2002. LNCS (LNAI), vol. 2431, pp. 348–360. Springer, Heidelberg (2002)
13. Mendes, P.N., Jakob, M., Bizer, C.: DBpedia for NLP: A Multilingual Cross-domain Knowledge Base. In: Proc. of the 8th International Conference on Language Resources and Evaluation, LREC 2012 (2012)
14. Miller, G.A.: WordNet: A Lexical Database for English. Communications of the ACM 38(11), 39–41 (1995)
15. OWL-S Coalition. OWL-based Web Service Ontology (2006), http://www.ai.sri.com/daml/services/owl-s/1.2/
16. Rahm, E., Bernstein, P.A.: A survey of approaches to automatic schema matching. The VLDB Journal 10(4), 334–350 (2001)
17. Sabou, M., d'Aquin, M., Motta, E.: SCARLET: SemantiC RelAtion DiscoveRy by Harvesting OnLinE OnTologies. In: Bechhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M. (eds.) ESWC 2008. LNCS, vol. 5021, pp. 854–858. Springer, Heidelberg (2008)
18. SAWSDL Working Group. Semantic Annotations for WSDL and XML Schema, SAWSDL (2007), http://www.w3.org/TR/2007/REC-sawsdl-20070828/
19. Shvaiko, P., Euzenat, J.: Ontology matching: State of the art and future challenges (2012)
20. Tarjan, R.E.: Data Structures and Network Algorithms, vol. 14. SIAM (1983)
21. Xing, Z., Stroulia, E.: UMLDiff: An Algorithm for Object-Oriented Design Differencing. In: Proc. of the 20th IEEE/ACM International Conference on Automated Software Engineering, pp. 54–65. ACM (2005)