A Formal Approach to Service Specification and Matching based on Graph Transformation

R. Heckel, A. Cherchago, M. Lohmann

Faculty of Computer Science, Electrical Engineering and Mathematics
University of Paderborn
Paderborn, Germany

Abstract

When Web services are composed by linking service providers and requestors, the requestor’s requirements for a “useful” service have to be matched against the service description offered by the provider. Among other things, service specifications (requirements or descriptions) may contain operation contracts specifying pre-conditions and effects of (required or provided) operations. In this paper we provide a semi-formal, UML-based notation for contracts and contract matching, as well as a formalization of these notions in terms of graph transformation. We establish the desired semantic relation between requestor and provider specifications and prove the soundness of our syntactic notion of matching w.r.t. this relation.

Keywords: Web services, contract matching, graph transformation

1 Introduction

The Internet, or more specifically the WWW, offers a virtually unlimited source of information and services to a skilled human user. We can translate texts, get information on the temperature in holiday resorts, book flights, and order travel guides without once leaving the computer. Even more amazing is the fact that we do not even have to know who will offer us these services. Knowing Google will usually suffice to quickly find offers for our needs. The ultimate vision of Web services is to transfer this ability to programs. These programs should be able to locate and invoke services at runtime over the net
to meet their own goals. This encompasses two problems: how to find these services and how to use them. In this paper, we will focus on the problem how to find them.

Current Web technologies already enable much of the discovery process: The interface of an offered service (usually a method to be invoked) can be specified in the Web Service Description Language (WSDL). This specification along with some keywords describing basic information about a Web service can be registered at a central UDDI-server. This registry serves as a central information broker and supplies information on possible service providers to the requests of clients.

Nevertheless, one very important question is still open: how a customer or, rather a customer application, can find an appropriate service that is compatible with the customer’s requirements? Nowadays this question is answered in a rather straightforward way: A developer queries a UDDI-server for a service, selects a suitable service from the results and integrates the invocation of this service into his program (this is supported e.g. by WebSphere Studio [6]). At runtime, the program can only try to execute the selected service. If this service is not available, e.g. due to network problems, the program cannot even automatically discover identical replacement services. This is quite far from the idea of dynamically discovering adequate services. Therefore, it is very important to establish techniques maximally automating the discovery process.

In our work the compatibility of provided and required services is defined via the compatibility between operations constituting the service interfaces. For all required operations it is necessary to find structurally and behaviorally compatible provided operations. The structural compatibility requires a correspondence between provided and required operation signatures. This can be checked using techniques developed for retrieving functions and components from software component libraries [12].

We concentrate on the second problem - behavioral compatibility. Service requestor and provider specify behavioral information about their operations by contracts [5]. We propose to use graph transformation rules for contract specification.

The classical interpretation of the rules, based on the double-pushout (DPO) approach to graph transformation [4], is not adequate for this purpose. It assumes that nothing is changed in the transformation beyond what is explicitly specified in the rule. An operation contract, however, represents a potentially incomplete specification of a transformation. Graph transitions have been proposed to provide a looser interpretation of graph transformation rules. The double-pullback (DPB) approach [7] defines graph transitions
and generalizes DPO by allowing additional changes, not encoded in the rule. This kind of the rule interpretation is adequate for rules specifying operation contracts.

Based on the notion of graph transitions we will define an operational understanding of what it means for a provider rule to match the requestor’s requirements. This shall be captured in a semantic matching relation. Since such a relation, being based on an infinite set of transitions, can not be computed directly, we introduce a syntactic matching relation which provides a sufficient condition for the semantic one.

After presenting in the next section the basic ideas of a service specification and sample application, in Section 3 we will discuss the issue of service specification matching, concentrating on the compatibility between provided and required operation contracts. The partial formalization of these notions, including the soundness of the syntactic relation w.r.t. the semantic one, will be given in Section 4. In Section 5 we conclude and list the open issues in the formalization.

2 Service Specification

In this section we consider the basic ingredients of service specifications and introduce a scenario of a Web service for a car rental system.

We start with the data model of the application expressed by the UML class diagram in Fig. 1: A rental company (class RentalCompany) owns vehicles (class Vehicle) of different types (classes Truck, Car, Van, Jeep). If a customer (class Customer) wants to rent a vehicle (association rents), it is necessary to reserve a vehicle (association reserves). Each legal entity, i.e. a customer or a company, has a name, an address and a bank account. The relation between a customer and a company is regulated by a contract (class EContract) containing all relevant renting information, such as a period of rent (class RentalInfo). To avoid additional complications, we assume that service requestor and provider are working with the same data model, agreed upon in advance.

"A Web service is an interface that describes a collection of operations that are network-accessible through standardized XML messaging" [9]. The next part of the service specification is represented by an interface. An example of the provided and required interfaces is presented in Fig. 2.

Interface contains structural information about operations while the behavior of these operations can be specified by contracts. The concept of contracts [5] is widely used within the Web services community to describe behavior of services and their constituents. A contract consists of a pre-condition speci-
fying the system state before some behavior is executed and a post-condition describing the system state after the execution of the behavior. There are different approaches employing formal techniques (e.g., description logic [11], situation calculus [10], algebraic specification languages [12], etc.) to contract specification. The main obstacle of these approaches is their lack of usability in the software industry, where knowledge and skill in the application of logic formalisms is scarce. Instead, we seek a notation that is close to the standard software modelling languages (e.g., UML) and has, at the same time, a formal background allowing to provide automation. This visual formal notation for contracts is provided by typed graph transformation [2].

In this context, a class diagram is considered as a directed attributed graph, whose vertices contain types and attribute declarations. Their relation with object diagrams representing run-time states is expressed by the notion of a type graph (TG) and corresponding instance graphs [2]. A graph transformation rule $p : L \Rightarrow R$ consists of a pair of $TG$-typed instance graphs $L,R$ with compatible structure, i.e., such that edges that appear in both $L$ and $R$ are connected to the same vertices in both graphs, vertices with the same name have the same type, etc. The left-hand side $L$ represents the pre-condition of the rule while the right-hand side $R$ describes the post-condition and effects (cf. the part of Fig. 3 marked by the dashed rectangle).

Sometimes, it is necessary to specify separately a context required for
the rule application. In this case a graph transformation rule with positive
application condition $\hat{p} : \hat{L} \supseteq L \Rightarrow R$ is used. In addition to $TG$-typed
instance graphs $L$ and $R$, $\hat{p}$ contains a graph $\hat{L}$ specifying an extension of $L$
by elements that are required for the application, but are not used otherwise.
(cf. Fig. 3).

Fig. 3 demonstrates the graph transformation rules for the provided operation
makeReserv and the required operation reservCar. $\hat{L}_p$ and $\hat{L}_r$ contain three
nodes representing input parameters of the operations: information about a
customer (vertices $c$ and $cus$), a rental period (vertices $ri$) and a hauling unit
(vertices $car$ and $my\_car$). The reservation is denoted by the edge reserves
connecting a customer with a car. This edge appears in $R_p$ and $R_r$ and represents
a result of the operation execution. The provided operation also creates the
vertex $ec\_EContract$ showing a contract that restricts availability of a car (edge
constructedBy).

The upper rule and the signature of the operation makeReserv represent
one possible variant of the reservation operation. Three other variants can be
obtained by simultaneous replacing the vertex with the type $Car$ in the rule and
the parameter with the type $Car$ in the signature by vertices and parameters
of the super-type $Vehicle$ (i.e. Jeep, Van, Truck). We can not use the type
Vehicle, because a car rental contract has to contain a precise specification of
the hauling unit.

To summarize, a service specification consists of a data model, structural
(operation signatures) and behavioral (operation contracts represented by
graph transformation rules) specifications of operations constituting a service.
In the next section we discuss service specification matching and consider an
example of matching required and provided operation contracts.
3 Specification Matching

In general, specification matching has to deal with all three aspects of a specification, i.e., data, signatures, and contracts. For simplicity, we ignore the matching of data models and discuss the matching of signatures only briefly (see [12] for a general discussion). As an example, we consider the relation between the required operation `reservCar` and the provided operation `makeReserv` whose signatures and contracts are depicted in Fig. 2 and Fig. 3 correspondingly.

The signatures of the operations differ for the result type that is present only in the provided operation. This does not violate compatibility because the output of the provided operation may simply be ignored by the requestor.

To determine the relation between signatures and contracts, we require that input and output parameters of each operation are represented by vertices with corresponding types in the rules. These dependencies are indicated by the dashed arrows in Fig. 3.

Now we consider behavioral compatibility which amounts to check compatibility of pre-conditions and effects. Pre-conditions are captured by positive application conditions \( \hat{L} \). In order to perform an operation successfully, the provider requires certain input data from the requestor as well as a certain properties to hold in the current states. In the provider rule of Fig. 3 this is information about a customer, a hauling unit supposed to be rent, and a period of renting. The requestor has to be prepared to deliver this data and to guarantee these properties. Hence the pre-condition of the requestor must entail the pre-condition of the provider, which is expressed by an occurrence (formally a graph homomorphism) from \( \hat{L}_p \) to \( \hat{L}_r \).

A requestor wants to have some benefit from the invocation of a service operation. If an operation does less then expected by a requestor, it is not considered to be useful. In other words, the effect of the provided operation must not be less than the effects specified by the requestor. That means, the requestor rule must be embedded in the provider as it is the case with the rules in Fig. 3. For example, the operation `makeReserv` additionally creates the vertex with the type `EContract` denoting an agreement between a company and a customer. This vertex is not presented in the requestor contract, because the goal of the requestor is to make a reservation, but not to sign a contract. Nevertheless, the effect of the provided operation fits the client requirements.

Next, we will present a (partial) formalization for the intuitive ideas obtained from the example.
4 Towards a Formalization

Contract matching can be formalized as a relation between graph transformation rules. In this section, we define two such relations, a semantic one based on the operational interpretation of rules, and a syntactic one which provides a sufficient condition for the semantic relation. First, however, we review some of the basic notions of the double-pushout (DPO) \cite{4} approach (see \cite{3} for a survey) and the double-pullback (DPB) approach \cite{7} to graph transformation.

4.1 The Double-Pushout Approach to Graph Transformation

Given a graph $TG$, called type graph, a $TG$-typed (instance) graph consists of a graph $G$ together with a typing homomorphism $g : G \rightarrow TG$ (cf. Fig. 4 on the left) associating with each vertex and edge $x$ of $G$ its type $g(x) = t$ in $TG$. In this case, we also write $x : t \in G$. A $TG$-typed graph morphism between two $TG$-typed instance graphs $(G, g)$ and $(H, h)$ is a graph morphism $f : G \rightarrow H$ which preserves types, that is, $h \circ f = g$.

The DPO approach to graph transformation has originally been developed for vertex- and edge-labelled graphs \cite{4}. Here, we present the typed version \cite{2}.

According to the DPO approach, graph transformation rules, also called graph productions, are specified by pairs of injective graph morphisms $(L \leftarrow K \rightarrow R)$, called rule spans. The left-hand side $L$ contains the items that must be present for an application of the rule, the right-hand side $R$ those that are present afterwards, and the context graph $K$ specifies the “gluing items”, i.e., the objects which are read during application, but are not consumed.

**Definition 4.1 (rule, graph transformation system)** A rule span typed over $TG$, in short $TG$-typed rule span, $s = (L \leftarrow K \rightarrow R)$ is a span of injective $TG$-typed graph morphisms.

A graph transformation system $GTS = (TG, P, \pi)$ consists of a type graph $TG$, a set of rule names $P$, and a mapping $\pi$ associating with each rule name $p$ a $TG$-typed rule span $\pi(p)$. If $p \in P$ is a rule name and $\pi(p) = s$, we say that $p : s$ is a rule of $GTS$.

A graph transformation rule with positive application condition $\hat{p}$ is a pair
Two examples of rules with positive application conditions are given in Fig. 3.

The transformation of graphs is defined by a pair of pushout diagrams, a so-called double-pushout construction.

**Definition 4.2 (DPO graph transformation)** A double-pushout (DPO) diagram \(d\) is a diagram as in Fig. 4 on the right, where (1) and (2) are pushouts. Given a type graph \(TG\) and a rule \(p : s\) with \(s = (L \xleftarrow{L} K \xrightarrow{R} R)\) the corresponding (DPO) transformation step from \(G\) to \(H\) is denoted by \(G \xrightarrow{p/d} H\), or simply \(G \xrightarrow{p} H\) if the diagram \(d\) is understood.

The span representation of the rule for the contract of the required operation \(\text{reservCar}\) (Fig. 3) and its application to an instance graph is given in Fig. 5.

Operationally speaking, the application of the rule proceeds as follows. Given the occurrence \(d_L\) of the left-hand-side \(L\) in \(G\), the application consists of two steps: The elements of \(G\) matched by \(L \setminus l(K)\) are removed, that does not change graph \(G\) in Fig. 5. Then, the elements matched by \(R \setminus r(K)\) are added to \(D\) which leads to the derived graph \(H\) additionally containing the \(\text{reserves}\) edge.

Gluing the graphs \(L\) and \(D\) over their common part \(K\) yields again the given graph \(G\), i.e., \(D\) is a so-called pushout complement and the left-hand square (1) is a pushout square. Only in this case the application is permitted. Similarly, the derived graph \(H\) is the gluing of \(D\) and \(R\) over \(K\), which forms...
the right-hand side pushout square (2).

This formalization implies that only vertices that are preserved can be merged or connected to edges in the context. It is reflected in the identification and the dangling condition of the DPO approach which characterize, given a rule $p : s = (L \leftarrow K \rightarrow R)$ and an occurrence $d_L : L \rightarrow G$ of the left-hand side, the existence of the pushout complement (1), and hence of a transformation step $G \xrightarrow{p/d} H$. The identification condition states that objects from the left-hand side may only be identified by the match if they also belong to the interface (and are thus preserved). The dangling condition ensures that the structure $D$ obtained by removing from $G$ all objects that are to be deleted is indeed a graph, that is, no edges are left "dangling" without source or target node.

This construction ensures that the changes to the given graph $H$ are exactly those specified by the rule. However, operation contracts represent specifications of operations that are, in general, incomplete, that is, additional effects should be allowed in the transformation. Therefore, a more liberal notion of rule application is required which ensures that at least the elements of $G$ matched by $L \setminus l(K)$ are removed, and at least the elements matched by $R \setminus r(K)$ are added. This kind of the rule interpretation is supported by the double-pullback (DPB) approach to graph transformation [7].

4.2 The Double-Pullback Approach to Graph Transformation

Graph transitions have been proposed to provide a looser interpretation of graph transformation rules. The double-pullback (DPB) approach introduces graph transitions and generalizes DPO by allowing additional changes, not encoded in the rule. Graph transitions are defined by replacing the double-pushout diagram of a transformation step with a double-pullback (DPB).

**Definition 4.3 (graph transition)** Let $p : s$ be a rule span with $s = (L \leftarrow K \rightarrow R)$. Then, a graph transition from $G$ to $H$ via $p$, denoted by $G \xrightarrow{\sim/d} H$, is a diagram like the right part of Fig. 6 where both (1) and (2) are pullback squares. A graph transition (or briefly transition) is called injective if both $g$ and $h$ are injective graph morphisms. It is called faithful if it is injective, and the morphisms $d_L$ and $d_R$ satisfy the following condition; for all $x, y \in L$, $y \notin l(K)$ implies $d_L(x) = d_L(y)$, and analogously for $d_R$.

A graph transition from $G$ to $H$ via a rule $\hat{p}$ with positive application condition, denoted by $G \xrightarrow{\hat{p}/d} H$, is a graph transition via a rule $p$, such that

\[ d_{L}(x) = d_{L}(y), \quad \text{and analogously for } d_{R}. \]
there exists $d_L$ satisfying $d_L = d_L \circ \hat{l}$ (cf. Fig. 6 on the left).

Notice that any pushout square of two given morphisms such that one of them is injective is also a pullback square. Thus, every DPO transformation is also a DPB transition.

Each faithful transition can be regarded as a transformation step plus a change-of-context [7]. This is modelled by additional deletion and creation of elements before and after the actual step. Faithful transitions capture our intuition about a loose interpretation of graph transformation rules for contract specification.

4.3 Semantic and Syntactic Matching

The notion of transition allows us to formalize semantically the desired notion of compatibility: Provider and requestor rules are semantically compatible if (i) applicability of the requestor rule implies applicability of the provider rule and (ii) every transition via the provider rule can be regarded as a transition via the requestor rule, too.

**Definition 4.4 (semantic matching)** Let $(p_1, \hat{L}_1)$ and $(p_2, \hat{L}_2)$ be graph transformation rules with positive application conditions, where $s_1 = (L_1 \xrightarrow{l_1} K_1 \xrightarrow{r_1} R_1)$ and $s_2 = (L_2 \xrightarrow{l_2} K_2 \xrightarrow{r_2} R_2)$. We say that $(p_1, \hat{L}_1)$ semantically matches $(p_2, \hat{L}_2)$, in symbols $(p_2, \hat{L}_2) \vdash_{\text{match}} (p_1, \hat{L}_1)$, iff

(i) for all graphs $G$, if there exists $d_{L_1} : \hat{L}_1 \to G$ such that $d_{L_1} := d_{L_1} \circ \hat{l}_1$ satisfies the identification condition of $p_1$, then there exists $d_{L_2} : \hat{L}_2 \to G$ such that $d_{L_2} := d_{L_2} \circ \hat{l}_1$ satisfies the identification condition of $p_2$, and

(ii) for all spans $t : (G \xrightarrow{g} D \xrightarrow{h} H)$, if there exists a transition $G \xrightarrow{p_2/d_2} H$, then there exists a transition $G \xrightarrow{p_1/d_1} H$ using the same bottom span $t$ (cf. Fig. 7).

This definition reflects the desired relation between contracts, but can hardly be applied for an algorithm determining contract compatibility. Therefore, we introduce a relation of *syntactic matching* that encompasses ideas presented in Section 3 and has more constructive character.
transformation rules with positive application conditions, where
We show that Def. 4.5 (i) / (ii) entails Def. 4.4 (i) / (ii),
Proof (Sketch)
matches with (Thm. 4.6 (soundness of matching)
Definition 4.5 (syntactic matching)
Let (p₁, Ł₁) and (p₂, Ł₂) be graph transformation rules with positive application conditions, where s₁ = (Ł₁ ↦ L₁)
K₁ r₁→ R₁) and s₂ = (Ł₂ ↦ L₂ K₂ r₂→ R₂). We say that (p₁, Ł₁) syntactically matches with (p₂, Ł₂), in symbols (p₂ : s₂, Ł₂) ⊨_match (p₁ : s₁, Ł₁), iff
(i) there exists an injective graph homomorphism h_L : Ł₂ → Ł₁ such that
h_L ◦  ł₂ satisfies the identification condition of p₂, and
(ii) there exist graph homomorphisms h_L : L₁ → L₂, h_K : K₁ → K₂, and
h_R : R₁ → R₂ such that the diagrams (a), (b), and the outer diagram in Fig. 7 on the left commute, and the diagrams (a) and (b) represent a faithful transition (cf. Fig. 7).

An example of syntactic matching is given in Section 3 for the graph transformation rules specifying the contracts of the required operation reservCar and the provided operation makeReserv.

Next, we demonstrate the soundness of our approach.

Theorem 4.6 (soundness of matching) Assume two graph transformation rules with positive application conditions ̂p₁ and ̂p₂. Then ̂p₂ ⊨_match ̂p₁ implies ̂p₂ ⊨_match ̂p₁.
Proof (Sketch) We show that Def. 4.5 (i) / (ii) entails Def. 4.4 (i) / (ii), respectively.
(i): Given d_L₁ : Ł₁ → G, we obtain d_L₂ : Ł₂ → G by d_L₁ ◦ h_L resulting in the commutativity of diagram (3). Morphism d_L₂ = d_L₂ ◦  ł₂ satisfies the identification condition of p₂ because of this commutativity and the fact that h_L ◦  ł₂ satisfies the identification condition of p₂. Moreover, we can show that the remaining diagrams in Fig. 7 on the left commute, thus relating the compatibility conditions for precondition and effect.
(ii): We have to show that for each faithful transition via the second rule there is a faithful transition via the first rule. By assumption, there exist graph homomorphisms between the first and the second rule (h_L, h_K, h_R), forming a faithful transition (cf. Fig. 7 on the right). Now, both transitions can be
vertically composed using the composition of the underlying pullback squares and faithfulness of the composed transition follows from the fact that the identification condition of $d_{L_1}$ follows from that of $h_L$ and $d_{L_2}$, and analogously for the right-hand side.

Completeness of syntactic matching requires a more refined relation at the semantic level, establishing a connection between statements (i) and (ii), that we have not yet fully worked out. The final section summarizes the main results and discusses more open problems.

5 Conclusion

In this paper we have developed formal concepts underlying a UML-based approach to service specification matching based on graph transformation rules for modelling operation contracts. We have used a loose interpretation of rules based on DPB graph transitions to obtain an operational understanding of contracts and a corresponding semantic matching relation, and we have established a syntactic relation providing a sufficient condition for the semantic one.

Several issues remain for future work. A sound and complete syntactic matching relation, requires a refinement of the semantic relation adding constraints on the compatibility between pre-conditions and effects. The formal presentation needs to be extended to typed graphs with attributes [8] and sub-typing [1], already used informally in the example.

The practical application of the theoretical concepts presented in our work is stipulated by finding an adequate XML-representation of contracts, and tool support for computing the syntactic matching relation.

References


