Structural and Behavioural Compatibility of Graphical Service Specifications ⋆,⋆⋆

R. Heckel [a] and A. Cherchago [b]

[a] *Department of Computer Science, University of Leicester, Leicester, LE1 7RH, United Kingdom*

[b] *International Graduate School "Dynamic Intelligent Systems", University of Paderborn, Warburger Str. 100, Paderborn, 33098, Germany*

Abstract

The ability of applications to dynamically discover required services is a key motivation for Web Services. However, this aspect is not entirely supported by current Web Services standards. It is our objective to develop a formal approach, allowing the automation of the discovery process. The approach is based on the matching of interface specifications of the required and provided services.

In the present paper, we establish an integral notion of structural and behavioural compatibility of service specifications. While structural information is represented by operation declarations, behavioural descriptions are provided by contracts expressed as graph transformation rules with positive and negative application conditions. The integration of structural and behavioural descriptions is facilitated by *typed and parameterised* graph transformation systems, augmenting the rule-based description of behaviour by a type graph and operation declarations representing the structural aspect.

The matching relation taking into account this combination is called *parameterised substitution morphism*. We show that substitution morphisms satisfy the semantic requirement inherent in its name: the substitutability of abstract operations by (calls to) concrete ones.

Key words: Web services, service specification matching, graph transformation, conditional parameterised rules, substitutability

⋆ Research funded in part by European Community's Human Potential Programme under contract HPRN-CT-2002-00275, [SegraVis].

⋆⋆Research funded in part by International Graduate School "Dynamic Intelligent Systems" of University of Paderborn.

Email addresses: reiko@mcs.le.ac.uk (R. Heckel), cherchago@upb.de (A. Cherchago).

1 Introduction

The Web Services platform provides the means to adopt the World Wide Web for application integration based on standards for communication, interface description, and discovery of services. The prosperity of this technology strongly depends on the ability of applications to discover useful services and select those that can be safely integrated with existing components. Much work has been done to achieve this aim. The interface of an offered service can be specified in the Web Service Description Language (WSDL). This specification along with some keywords characterising the service can be published at a UDDI-registry which serves as a central information broker and supplies this information to potential clients. However, current standards do not support the automation of checking behavioural compatibility of the requestor's requirements with service descriptions.

The academic community has proposed a number of approaches describing the behaviour of services by *contracts* [1], usually expressed by means of pre- and post-conditions in some form of logic (see Section 4 for a discussion). The main obstacle of this idea is its lack of usability by professional software engineers, whose skills in applying logic formalisms are scarce. In [2–4] it has been observed that graph transformation rules could provide a more abstract, visual representation of contracts, specifying preconditions and effects of operations by graphical patterns. This has the advantage of blending intuitively into standard modelling techniques like the UML, providing contracts with an operational (rather than purely descriptive) flavour.

However, the use of the graph transformation technique for Web service contracts is hampered by two problems. Firstly, the classical semantics of rules (as formalised, for example by the double-pushout (DPO) approach to graph transformation [5]) assumes that transformations are completely described by rules, i.e., nothing changes beyond what is explicitly required. A contract, however, represents a potentially incomplete specification of an operation, rather than its implementation, and the strict rule semantics does not reflect the loose nature of contracts. Secondly, classical graph transformation rules do not contain information on the signatures of the specified operations, like input and output parameter types needed to ensure type safety in the interaction.

For the first problem, we propose to use the loose semantics of rules based on *graph transitions*. This kind of semantics has been formalised in the double-pullback (DPB) approach [6] which defines graph transitions as a generalisation of DPO transformations, allowing changes that are not required by the rules. In order to increase the expressiveness of our graphical contract language, we consider rules with positive and negative application conditions. Negative conditions are well-known to increase the expressive power

of rules [7]. In the classic approaches, positive application conditions can be encoded by extending both the left- and the right-hand side of a rule by the required elements: they become part of the context. This is no longer possible, in the presence of unspecified effects. In fact, the implicit frame condition, that all elements present before the application and not explicitly deleted, are still present afterwards, is no longer true. Thus, an element matched by a positive application condition may disappear, while an element which is shared between the left- and the right-hand side must be preserved.

Since a service may contain several operations, contracts represented by conditional rules are collected in *conditional graph transformation systems* (GTS) providing the *behavioural* specification of the entire service. All conditional rules of GTS are constructed over a conceptual data type model, represented as a type graph.

To tackle the second problem, the *structural* description of a service is given by a *signature* similar to those known from algebraic specifications [8]. A signature comprises a type graph and operation declarations. An *integral* service description is obtained by an amalgamation of the structural and behavioural specifications, i.e., service signature and conditional GTS, and called *conditional parameterised GTS*. Each rule of such system is equipped with the parameters corresponding to a declaration of the specified operation.

In our work a *compatibility* of provided and required services is defined via the compatibility of operations constituting the service interfaces: For all required operations it is necessary to find structurally and behaviourally compatible provided operations. Structural compatibility requires a correspondence between service signatures, which is captured by a *signature morphism*. Behavioural compatibility amounts to relate the required and provided conditional GTSs. We establish semantic requirements underlying the contract resemblance and use them for the construction of an appropriate relation. This relation, formally defined by a *substitution morphism*, allows to match service specifications developed in different type contexts, which is the first main contribution of the paper. We demonstrate that substitution morphisms satisfy the semantic requirements.

In our previous works (cf. [2,3]), structural and behavioural aspects of compatibility have been considered separately, the current presentation introduces the notion of integral compatibility absorbing the two compatibility aspects. The intended correspondence between integral specifications of services is modelled by a *parameterised substitution morphism*.

The rest of the paper is organised as follows: After presenting in the next section the constituents of service specifications and their formalisation in terms of graph transformation, specification matching is treated in Section 3.

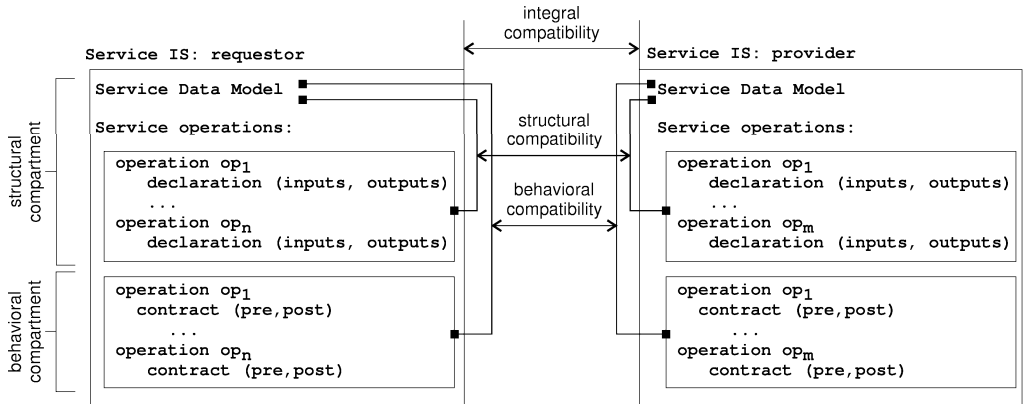

Fig. 1. Service interface specifications and their compatibility.

In Section 4 we discuss related approaches, and in Section 5 we summarise the main achievements and list issues for future work.

2 Service Interface Specification

In this section we consider the basic ingredients of service specifications by means of an example from the travel business domain. We discuss two sample specifications reflecting the views of service provider and requestor of a Web service for booking hotel rooms. The scenario is not complete, but it is in line with standardisation efforts in the travel industry [9].

The overall structure of service specifications is illustrated in Fig. 1. We distinguish between requestor and provider specifications. Each service specification consists of two compartments, a structural and a behavioural one. The *structural compartment* is composed of a *data model* and *operation declarations*. This information is typical for interface descriptions of components as supported by conventional middleware platforms (see, e.g., [10]). A *behavioural compartment* comprises semantic descriptions of the service operations in terms of *contracts* [11].

The structural compartment of interface specifications is considered in more detail in the next subsection, while Subsection 2.2 introduces the behavioural compartment containing operation contracts represented by transformation rules and formalised by conditional graph transformation systems. An integration of the structural and behavioural compartments is presented in Subsection 2.3, where the entire service interface is captured by conditional parameterised graph transformation systems.

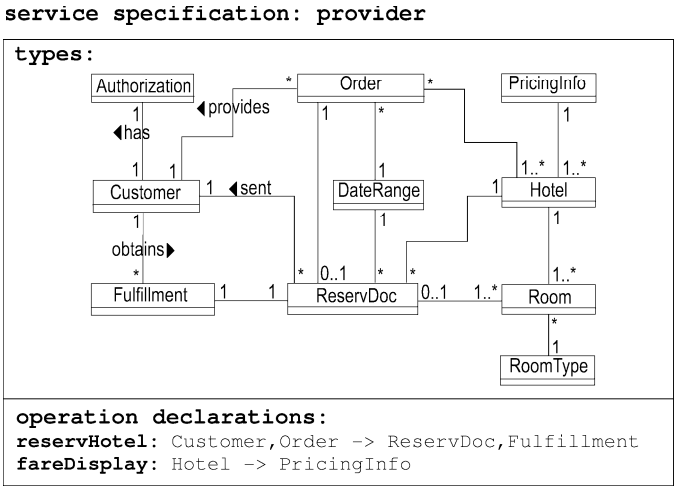

Fig. 2. Structural compartment of the provided interface specification: data model (top) and operation declarations (bottom).

2.1 Structural Specification

The structural compartment consists of data types and operation declarations.

2.1.1 Data Types

As already mentioned, the first ingredient of the structural compartment in the required and provided interface specifications is a data model, corresponding to an ontology in the context of the Semantic Web [12]. In general, the data model describes the concepts and relations shared across a family of applications or an entire business domain.

Example 2.1. The structural compartment of the provided interface specification for the hotel reservation service is shown in Fig. 2. The upper part of this figure represents a fragment of the data model whose constituents are interpreted as follows: A customer (type Customer) intends to book a room (type Room) of a specific kind (type RoomType) in a hotel (type Hotel). A booking process whose execution may be allowed only for some specific group of authorized users (type Authorization) is initiated by an order submission (type Order) containing besides other information a reservation time slot (type DateRange). A successful completion of booking is indicated by a reservation document (type ReservDoc) and payment information being relevant to the booking (type Fulfillment). The latter is derived from a price (type PricingInfo) associated with each room.

The structural compartment of the required interface specification is given in Fig. 3. In general, the required data model depicted in the left-hand side of the figure is quite similar to the provided one. However, some semantically iden-

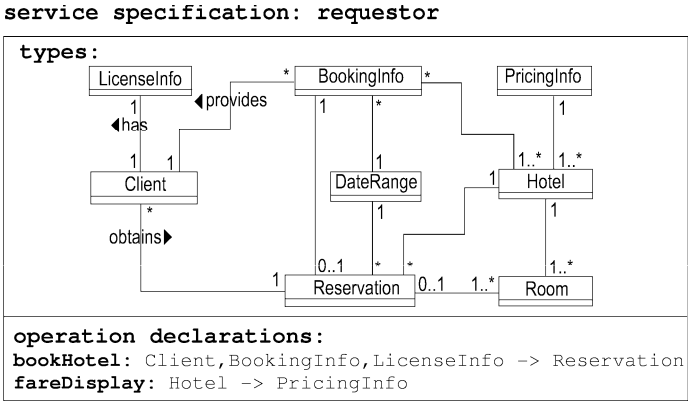

Fig. 3. Structural compartment of the required interface specification: data model (top) and operation declarations (bottom).

tical types in the models have different identifiers, because the required and provided systems are developed independently from each other. For example, the types BookingInfo, Client, and LicenseInfo from the required data model appear in the provided one under the names Order, Customer, and Authorization. △

In order to match required and provided specifications automatically, we need to find an appropriate formalism for each constituent of the interface specification. The formal description of structural information is based on typed graphs.

The distinction of types and instances, as it occurs in data modelling (ER diagrams and instances) or object-oriented programming (classes and objects) is formally captured by the notion of *type and instance graphs*.

By *graphs* we mean directed unlabelled graphs $G = \langle G_V, G_E, src^G, tar^G \rangle$ with set of vertices G_V, set of edges G_E, and functions $src^G : G_E \to G_V$ and $tar^G : G_E \to G_V$ associating with each edge its source and target vertex. A graph homomorphism $f : G \to H$ is a pair of functions $\langle f_V : G_V \to H_V, f_E : G_E \to H_E \rangle$ preserving source and target, that is, $src^H \circ f_E = f_V \circ src^G$ and $tar^H \circ f_E = f_V \circ tar^G$. With componentwise identities and composition this defines the category **Graph**.

Vertices and edges of the *type graph* represent types. Each type graph gives rise to a number of instance graphs representing states or patterns that will be relevant in the formalisation of the behavioural compartment. Given a type graph TG, a *TG-typed (instance) graph* consists of a graph G together with a typing homomorphism $g : G \to TG$ 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 $\langle G, g \rangle$ and $\langle H, h \rangle$ is a graph morphism $f : G \to H$ which preserves types, that is,

$h \circ f = g$. With composition and identities this defines the category $\mathbf{Graph}_{TG}$, which is the comma category of **Graph** over TG.

2.1.2 Operation Declarations

In addition to data types, the structural compartment of service interfaces provides a set of operation declarations.

Example 2.2. Declarations of provided operations for the hotel reservation service are shown in the lower part of Fig. 2. The operation reservHotel requires a caller to submit a customer profile and a booking order, and yields a reservation document together with a financial fulfillment. The operation fareDisplay obtains a hotel identifier as an input and returns hotel fairs. This operation is implemented in such a way that its execution is not accompanied with an inventory check for available places in a hotel whose fares are displayed.

The required operation declarations are depicted in the lower part of Fig. 3. While the declaration of the operation bookHotel is slightly different from the one of the provided operation reservHotel, their intended meaning coincides. The declaration of the operation fareDisplay is identical to the corresponding provided operation. However, the requestor intends to find an operation which displays only the fares of hotels available for booking . △

Each operation $p : v \to w$ is identified by a name p and has associated lists of input and output parameter types $v = v_1, \ldots, v_n$ and $w = w_1, \ldots, w_m$, respectively, with $w_i, v_j \in TG_V \cup TG_E$. A type graph with operation declarations represents the structural description of a service, modelled by a *service signature*.

Definition 2.3 (service signature). A *service signature* $\mathcal{S}$ is a pair $\langle TG, P \rangle$ consisting of a type graph TG and a family of sets $P = (P_{v,w})_{v,w \in |TG|^*}$ with operation declarations of the form $p : v \to w$ for $p \in P_{v,w}$. △

As discussed above, operations with identical declarations may provide different behaviours (cf. operations fareDisplay in Example 2.2). If the parties are not aware of this fact, interaction problems may occur. To avoid such problems, the structural descriptions of the operations have to be accompanied with *behavioural* information expressing the intended meaning of the operations. In our approach, the behaviour of operations is specified by contracts.

2.2 Behavioural Specification

Contracts were originally introduced by Bertrand Meyer in the object-oriented design approach called Design by Contract [11]. An operation contract consisting of *pre- and post-conditions* is used to ensure the correctness of interaction between a supplier of operation and clients calling the operation. The pre-condition describes the conditions which must be fulfilled prior to operation invocation. The post-condition portrays the effect of the operation, i.e., state changes that occur when the operation completes successfully. Typically, pre- and post-conditions are assertions, e.g., Boolean expressions, stating some properties of the entities in the system states.

There are different approaches to contract specification based on description logics [13], algebraic specification languages [14], etc. As already mentioned, such logic-based approaches are difficult to use and integrate into mainstream software development approaches. Therefore, we aim at a notation that is close to standard software modeling languages and that has at the same time a formal semantics to allow for automation. This visual formal notation for contracts is provided by *typed graph transformation* [15].

2.2.1 Typed Graph Transformation

Representing runtime states as typed graphs, state changing operations are described by *graph transformation rules* (or productions) $p : s$ consisting of a rule name p, and a span $s = (L \supseteq K \subseteq R)$, where L, K and R are TG-typed instance graphs. The left-hand side L and the right-hand side R describe a part of the system state before and after execution of the operation, that is, the pre- and postconditions, and the context graph K contains those elements that are read but not deleted by the operation. For a rule $p : s$ we usually assume that the graph K is the intersection $K = L \cap R$. In this case, we denote the rule by $p : L \to R$.

Example 2.4. Graph transformation rules specifying the required operation bookHotel and the provided operation reservHotel are shown in Fig. 4. They represent the first attempt at describing behaviour of the corresponding operations and will be refined latter. The left-hand sides of the rules contain elements standing for the input parameters of the operations and their relationships. For example, the edges between the objects c:Client and bi:BookingInfo in the requestor rule, and the objects cus:Customer and o:Order in the provider rule denote the fact that the booking request is submitted by a specific customer. The output parameters of the operations appear in the right-hand sides of the rules. △

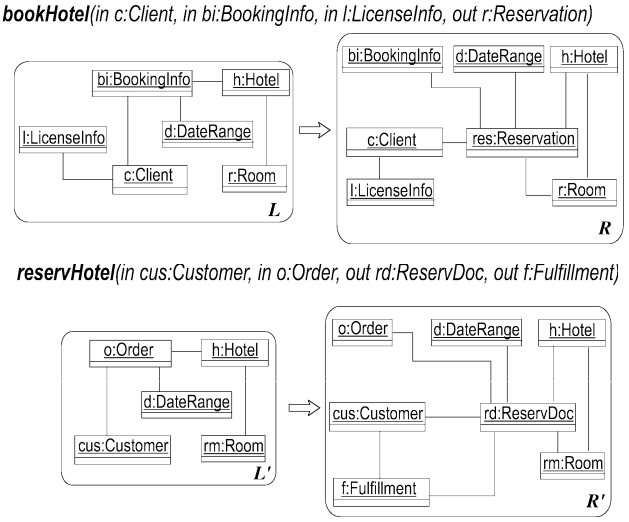

Fig. 4. Graph transformation rules bookHotel (top) and reservHotel (bottom).

Fig. 5. Typed span morphism (left) and double-pushout (or -pullback) diagram (right).

Formally, graph transformation rules are specified by spans of injective graph morphisms $L \xleftarrow{l} K \xrightarrow{r} R$ as shown in the following definition. While the behaviour of a single operation is described by a graph transformation rule, several behavioural descriptions can be aggregated in a (typed) graph transformation system (GTS).

Definition 2.5 (typed span, graph transformation system). A *TG-typed span* s is an expression of the form $(L \xleftarrow{l} K \xrightarrow{r} R)$, where l and r are injective TG-typed graph morphisms.

Given TG-typed spans $s = (L \xleftarrow{l} K \xrightarrow{r} R)$ and $s' = (L' \xleftarrow{l'} K' \xrightarrow{r'} R')$, a *$TG$-typed span morphism* $e : s \to s'$ is a tuple $\langle e_L, e_K, e_R \rangle$ of TG-typed graph morphisms commuting with l, l', r and r' (cf. Fig. 5 on the left).

A (typed) *graph transformation system* $\mathcal{G} = \langle TG, P, \pi \rangle$ consists of a *type graph* TG, a set of rule names P, and a mapping π associating with each rule name p a TG-typed span $\pi(p)$. If $p \in P$ is a rule name and $\pi(p) = s$, we say that $p : s$ is a rule of $\mathcal{G}$. △

A change from state G to state H under the execution of an operation specified by rule $p : L \to R$ is modelled by a graph transformation step $G \overset{p}{\Longrightarrow} H$. This requires that a renaming of L occurs as a subgraph in G. Then, $L \setminus R$ (which

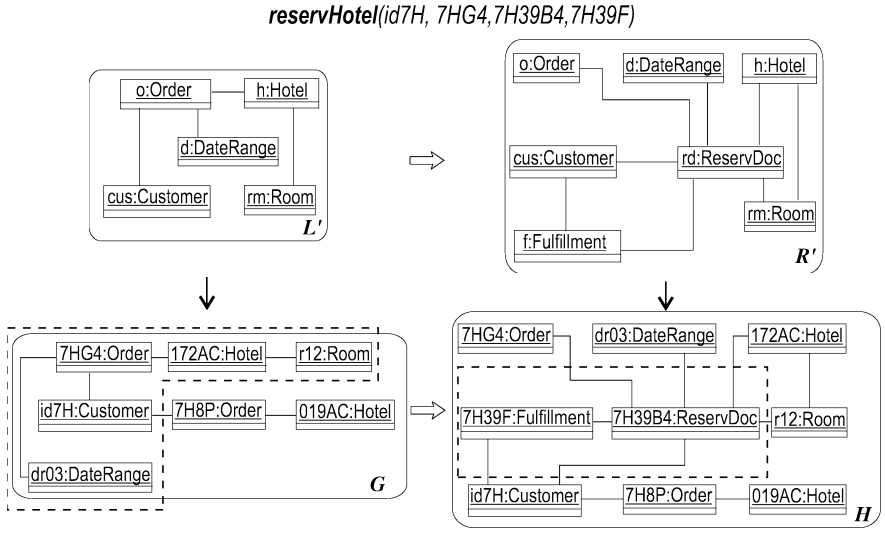

Fig. 6. A sample transformation step via the rule reservHotel.

consists of all nodes and edges of L not belonging to R) is removed from G, and $R \setminus L$ is added to the result. This leads to the derived graph H which contains a copy of R as a subgraph.

The effect encoded in the rule is defined by the elements which have to be deleted (exist only in L), created (exist only in R), and preserved (exist in K) under the rule application. The deleted and created elements are denoted by $del(p)$ and $add(p)$, accordingly.

The semantics described above is called *strict*, because the application deletes and creates exactly what is specified by the rule. An implicit *frame condition* ensures that the remaining part of the graph is left unchanged.

Example 2.6. Fig. 6 demonstrates a transformation step via the rule reservHotel. First of all, we look for an occurrence of the left-hand side L' of the rule in the instance graph representing (a fragment of) the system state. This occurrence is marked by the dashed polygon in the source graph G. Then, the elements matching $del(reservHotel)$, i.e., the edges between the object 7HG4:Order and the objects dr03:DateRange, id7H:Customer, and 172AC:Hotel are deleted, and the elements corresponding to $add(reservHotel)$ marked by the dashed rectangle in the target graph H are added to the result. The newly created objects 7H39B4:ReservDoc and 7H39F:Fulfillment are obtained as the output of the provided operation specified by the rule reservHotel. △

The strict semantics is formalised by the *double-pushout* (DPO) approach to graph transformation [16], where transformations of graphs are defined by pairs of pushout diagrams, called double-pushout constructions (cf. Fig. 5 on the right).

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

Definition 2.7 (DPO graph transformation). Given a graph transformation system $\mathcal{G} = \langle TG, P, \pi \rangle$ and a rule $p : (L \xleftarrow{l} K \xrightarrow{r} R)$, a *(DPO) transformation step* in $\mathcal{G}$ from G to H via p, denoted by $G \overset{p/d}{\Longrightarrow} H$, is a diagram d like in the right of Fig. 5, where both (1) and (2) are pushout squares. We also write p/d if G and H are understood, and denote by $top(d)$ and $bot(d)$ the top and bottom span of d.

A *transformation sequence* $\rho = \rho_1 \dots \rho_n : G \Rightarrow^* H$ in $\mathcal{G}$ via $p_1, \dots, p_n$ is a sequence of transformation steps $\rho_i = (G_i \overset{p_i/d_i}{\Longrightarrow} H_i)$ such that $G_1 = G, H_n = H$ and consecutive steps are composable, that is, $G_{i+1} = H_i$ for all $1 \leq i < n$.

The *category of transformation sequences over* $\mathcal{G}$ denoted by $\mathbf{Trf}(\mathcal{G})$ has all graphs $G \in \mathbf{Graph}_{TG}$ as objects and all transformation sequences in $\mathcal{G}$ as arrows. △

The existence of the pushout complement (1), and hence of a direct derivation[1] $G \overset{p/d}{\Longrightarrow} H$, is characterized by the *gluing conditions* [17]: The *dangling condition* ensures that the structure D obtained by removing from G all objects to be deleted is indeed a graph, that is, no edges are left "dangling" without source or target node. The *identification condition* states that objects from the left-hand side may only be identified by the match if they also belong to the context graph K (and are thus preserved).

2.2.2 Loose Semantics of Rules

The strict semantics is pertinent for the contracts of the provider, who obviously has complete information on supplied functionality. The required contracts, in turn, are incomplete specifications of service behaviour, because a developer of the requestor system has only loose idea of the provided services. The loose contracts describe minimally required effects, allowed to be extended in more powerful provided operations.

Therefore, the contract rules of the required operations have to be interpreted in a more liberal way: *at least* the elements of the graph G matched by $del(p)$ are removed, and *at least* the elements matched by $add(p)$ are added. This

[1] The pushout (2) always exists since the category $\mathbf{Graph}_{TG}$ is cocomplete.

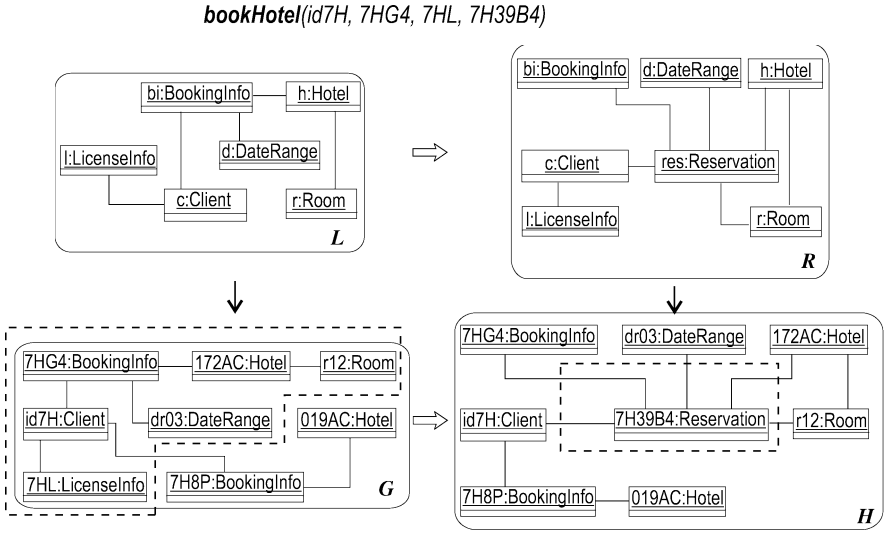

Fig. 7. A sample graph transition via the rule bookHotel.

semantics of rules, called *loose*, is provided by the notion of *graph transition* which leaves out the frame condition.

Like a transformation step, a graph transition $G \overset{p/d}{\rightsquigarrow} H$ from G to H via p requires that L occurs in G and carries out the transformations of G being explicitly encoded in the rule, but there may be unspecified deletions and additions as well.

Example 2.8. A sample graph transition is shown in Fig. 7. It applies the rule bookHotel while in parallel the object 7HL:LicenseInfo and the edge connecting this object with the object id7H:Client is deleted. The "spontaneous" deletion illustrates an effect which is unspecified by the rule bookHotel. This unspecified effect may be imposed by some compatible provided operation extending the effect defined in the rule bookHotel. △

The more liberal notion of rule application discussed above is provided by the double-pullback (DPB) approach to graph transformation [6]. The DPB approach introduces *graph transitions* and generalizes DPO by allowing additional, unspecified changes. Formally, graph transitions are defined by replacing the double-pushout diagram of a transformation step with a double-*pullback*.

Definition 2.9 (DPB graph transitions). Given a graph transformation system $\mathcal{G} = \langle TG, P, \pi \rangle$ and a rule $p : (L \xleftarrow{l} K \xrightarrow{r} R)$, a *transition* in $\mathcal{G}$ from G to H via p, denoted by $G \overset{p/d}{\rightsquigarrow} H$, is a diagram like in the right of Fig. 5, where both (1) and (2) are pullback squares.

A 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 $d_L(x) \neq d_L(y)$, and analogously for d_R.

A *transition sequence* $\rho = \rho_1 \dots \rho_n : G \rightsquigarrow^* H$ in $\mathcal{G}$ via $p_1, \dots, p_n$ is a sequence of faithful transitions $\rho_i = G_i \overset{p_i/d_i}{\rightsquigarrow} H_i$ such that $G_1 = G, H_n = H$ and consecutive steps are composable, that is, $G_{i+1} = H_i$ for all $1 \leq i < n$.

The *category of transitions over* $\mathcal{G}$, denoted by $\mathbf{Trs}(\mathcal{G})$, has all graphs $G \in \mathbf{Graph}_{TG}$ as objects and all transition sequences in $\mathcal{G}$ as arrows. $\triangle$

The condition ensuring the faithfulness of transitions means that d_L and d_R satisfy the identification condition of the DPO approach with respect to l and r. 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, in turn, can be regarded as a transformation step plus a change-of-context [6]. This is modeled by additional deletion and creation of elements before and after the actual step. In the following we stick in our presentation to the faithful transitions.

2.2.3 Conditional Graph Transformation

As we will see in the next section, behavioural compatibility between the required and provided operations is ensured via a matching of preconditions and effects of rules. For this to be possible, the two parts of the rules have to be clearly separated. However, normally the specification of the precondition is mixed up with that of items to be deleted in the left-hand side of the rule.

A refinement of the rule's structure avoiding this problem is obtained by using *positive application constraints* in the form $L \subseteq \hat{L}_P$, where L is the left-hand side of a rule span and $\hat{L}_P$ is a *positive precondition pattern*. The elements constituting $\hat{L}_P$ compose a context required to be present for the rule application.

Note that, in the case of the strict semantics, the use of positive constraints does not increase the expressiveness of rules: They can be integrated by extending both the left- and the right-hand side with the elements required, but not deleted by the rule. This is correct because we are sure that everything that is not explicitly deleted is not deleted at all.

This is no longer true under the loose semantics, where the positive precondition may extend the left-hand side of the rule with elements that are required to be there, but which may be deleted spontaneously, without explicit specification.

Positive constraints allow us to assert the *existence* of patterns in graphs.

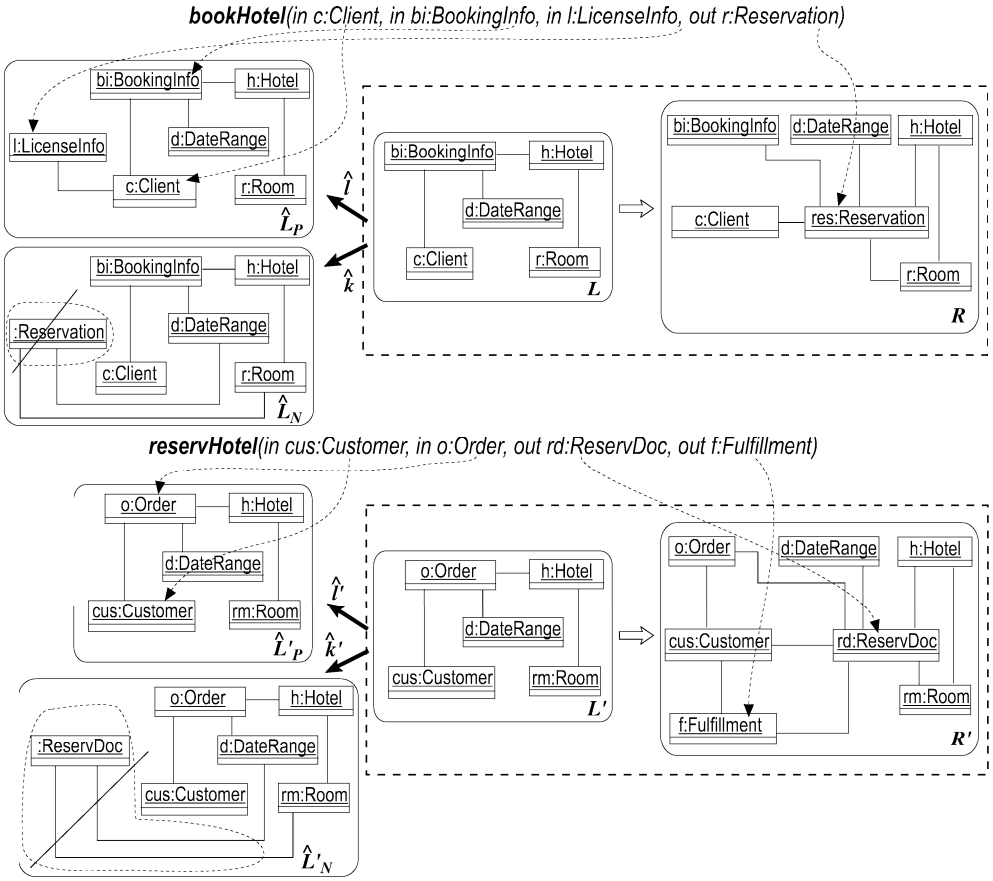

Fig. 8. Contract rules for the operations bookHotel (top) and reservHotel (bottom).

However, the language introduced so far does not allow to specify the *non-existence* of patterns. To fill this gap, the expressive power of the contract language is increased by *negative application constraints* $L \subseteq \hat{L}_N$, where $\hat{L}_N$, called *negative precondition pattern*, extends L with the elements that must not be present in a graph when the rule is applied.

Example 2.10. Fig. 8 shows conditional rules defining the contracts of the required operation bookHotel and the provided operation reservHotel. The parts of Fig. 8 marked by the dashed rectangles portray the effects expected by the requestor and guaranteed by the provider, accordingly. The output parameters of the operations appear in the right-hand sides of the rules. $\hat{L}_P$ and $\hat{L}'_P$ specify the positive precondition patterns containing the input parameters of the operations.

Besides customer and order information being present in the positive precondition patterns of both rules, $\hat{L}_P$ additionally contains the business license code of the travel agency making the reservation. The requestor considers the parameter l:LicenseInfo as the input which may be expected by the service provider. At the same time, this parameter is not needed for the following computations in the requestor system, so it appears only in the positive pre-

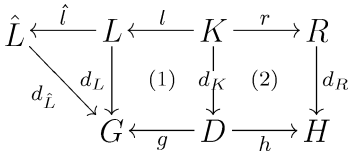

Fig. 9. Conditional span.

condition pattern and its further behaviour is left unspecified, i.e., it can be unrestrictedly manipulated by the provider. The negative precondition patterns $\hat{L}_N$ and $\hat{L}'_N$ of the rules prevent us from booking a room which has been already reserved. △

The positive and negative constraints compose an *application condition* over a rule span. A rule span enriched with such application condition, called conditional span, is formally introduced in the following definition.

Definition 2.11 (conditional span). An *application condition* $A(s) = \langle AP(s), AN(s) \rangle$ over a TG-typed span $s = (L \xleftarrow{l} K \xrightarrow{r} R)$ consists of two sets of injective typed graph morphisms $AP(s), AN(s)$ outgoing from L which contain positive and negative constraints, respectively. $A(s)$ is called positive (negative) if $AN(s)$ ($AP(s)$) is empty.

Let $L \xrightarrow{\hat{l}} \hat{L}$ be a positive or negative constraint and $L \xrightarrow{d_L} G$ be a typed graph morphism (cf. Fig. 9). Then d_L *P-satisfies* $\hat{l}$, if there exists a typed graph morphism $\hat{L} \xrightarrow{d_{\hat{L}}} G$ such that $d_{\hat{L}} \circ \hat{l} = d_L$. d_L *N-satisfies* $\hat{l}$, if it does not *P-satisfy* $\hat{l}$.

Let $A(s) = \langle AP(s), AN(s) \rangle$ be an application condition and $L \xrightarrow{d_L} G$ be a typed graph morphism. Then d_L *satisfies* $A(s)$, if it *P-satisfies* at least one positive constraint and *N-satisfies* all negative constraints from $A(s)$.

A *conditional span* is an expression of the form s if $A(s)$, where $s = (L \xleftarrow{l} K \xrightarrow{r} R)$ is a TG-typed span, $A(s)$ is an application condition over s. It is *applicable* to a graph G via $L \xrightarrow{d_L} G$ if d_L satisfies $A(s)$. △

Notice that positive application conditions consist of a disjunction of positive constraints, in contrast with the conjunction in [7]. That means, positive and negative conditions are, in fact, dual to each other.

Having defined conditional spans, we turn to graph transformation systems with conditional rules being employed as a formal specification of the behavioural compartment.

Here, a transformation step or transition via a conditional rule requires a match of L within G. Then, it is necessary to check whether occurrences of the positive (negative) precondition patterns corresponding to the chosen

match are contained (are not contained) in the source graph G. If it is the case, we say that the match of L in G satisfies the application condition and proceed similarly to the unconditional case. Otherwise application of the rule at the chosen match is forbidden.

Definition 2.12 (conditional GTS). A *conditional graph transformation system* $\mathcal{C} = \langle TG, P, \pi \rangle$ consists of a type graph TG, a set of rule names P, and a mapping π providing for each rule name p a TG-typed conditional span $\pi(p)$. If $p \in P$ is a rule name and $\pi(p) = s \text{ if } A(s)$, we say that $p : s \text{ if } A(s)$ is a conditional rule of $\mathcal{C}$.

Given a conditional rule $p : s \text{ if } A(s)$ of $\mathcal{C}$, then a *transformation step* (*transition*) in $\mathcal{C}$ from G to H via p is a transformation step (transition) via an unconditional rule $p : s$ such that $d_L \in d$ satisfies $A(s)$ (cf. Fig. 9) $\triangle$

For a rule $p : s \text{ if } A(s)$ with $A(s) = \langle AP(s), AN(s) \rangle$ we assume that the injective morphisms $L \xrightarrow{\hat{l}} \hat{L}$ representing positive or negative constraints are indeed inclusions, and that the pairwise intersection of all the $\hat{L}$ is well-defined and equal to L. That means, the left-hand side of a rule and all its constraints are defined over a common name space, a fact that will be relevant when introducing parameterised rules.

Structural and behavioural compartments of a service interface specification, i.e., service signature $\mathcal{S} = \langle TG, P \rangle$ and conditional graph transformation system $\mathcal{C} = \langle TG, P, \pi \rangle$, are developed so far independently from each other. In the following, we relate them in order to get an integral description of a service interface.

2.3 Integration of Structural and Behavioural Compartments

An integration of the two specification compartments basically requires to combine operation declarations and conditional rules describing the behaviour of these operations. An operation declaration specifies types of input and output parameters appearing as elements in the corresponding rule, i.e., in the positive precondition patterns $\hat{L}^i_P$, and in the right-hand side R of the rule span. The desired integration is captured by the notion of *conditional parameterised rules*. The names of such rules are given by *parameter expressions* of the form $p(x, y)$, where $p : v \to w$ is an operation declaration, and $x = x_1, \ldots, x_n$ and $y = y_1, \ldots, y_m$ are sequences of input and output parameters conforming to the types constituting v and w.

Definition 2.13 (conditional parameterised rule). Given a signature $\mathcal{S} = \langle TG, P \rangle$, and a pair of TG-typed graphs $\langle X, Y \rangle$, a set of *parameter*

expressions over $\mathcal{S}$ *and* $\langle X, Y\rangle$ is defined by $\mathcal{E}_P(X,Y) = \{p(x,y) | p \in P_{v,w}, x \in X_v, y \in Y_w\}$.

A *conditional parameterised rule cp* over $\mathcal{S} = \langle TG, P\rangle$ is an expression of the form $p(x,y) : s \text{ if } A(s)$, where $s \text{ if } A(s)$ is a TG-typed conditional span, and $p(x,y) \in \mathcal{E}_P(\mathcal{L}, R)$ is a parameter expression over $\langle \mathcal{L}, R\rangle$, i.e, $x \in \mathcal{L}_v$ and $y \in R_w$, for $p \in P_{v,w}$ and $\mathcal{L} = \bigcup \hat{L}^i_P$, assuming that the union $\mathcal{L}$ of all positive precondition patterns is well-defined. △

A transformation step or transition via a conditional parameterised rule is analogous to the one via the ordinary rule, however it can be additionally annotated with the elements from the source and target graphs matching the declared parameters (cf. Fig. 6 and Fig. 7).

Now we are able to aggregate service signature $\mathcal{S} = \langle TG, P\rangle$ and conditional graph transformation system $\mathcal{C} = \langle TG, P, \pi\rangle$ in order to get a formal specification of the entire service interface.

Definition 2.14 (conditional parameterised GTS). A *conditional parameterised graph transformation system* $\mathcal{CP} = \langle TG, P, CP\rangle$ is a triple, where TG and P compose a signature, and CP is a set of conditional parameterised rules over this signature, such that $p(x,y) : s_i \text{ if } A(s_i)$ for $i = 1, 2$ implies $s_1 \text{ if } A(s_1) = s_2 \text{ if } A(s_2)$. △

The definition states that each rule in $\mathcal{CP}$ has unique name, so the conditional parameterised GTS can be represented by the conditional GTS $\mathcal{C} = \langle TG, P, \pi\rangle$, where P contains parameter expressions for each rule from CP, and π associates $p(x,y)$ with $s \text{ if } A(s)$ if and only if the corresponding rule appears in CP. Hence transformations or transitions in $\mathcal{CP}$ are similar to the ones in the graph transformation system $\mathcal{C}$.

Next, we discuss the compatibility between the required and provided interface specifications containing the client's requirements for a useful service and service descriptions.

3 Compatibility of Service Interface Specifications

The notion of compatibility is motivated by a *substitution principle*: a replacement of abstract operation descriptions in the required system with the concrete operations implemented in the provided system should guarantee that the behaviour of integrated system remains acceptable for the parties.

The structural aspect of compatibility (cf. Fig 1) is formalised in the next subsection by a signature morphism associating the structural specifications

of the two parties. Subsection 3.2 discusses and formalises behavioural compatibility as a substitution morphism between the conditional graph transformation systems. Signature and substitution morphisms are aggregated in Subsection 3.3, where we establish the integral compatibility in terms of a parameterised substitution morphism.

3.1 Structural Compatibility

Structural compatibility between required and provided interface specifications is captured by matching service signatures which may be developed in the context of different data models. In order to collate the signatures of the parties, we have to provide an appropriate retyping which makes operation declarations comparable with each other. The retyping is based on a mapping between the required and provided type graphs representing the data models.

Due to the potential heterogeneity of service-oriented systems, their interface specifications and, in particular, data models may arbitrarily diverge. To compute a semantically meaningful relation automatically is a problem that can hardly be solved in general. In practice, requestor and provider either construct their specifications over a common standard ontology or use non-standard models equipped with mappings to a standard one. An example of a standard ontology for the travel business domain can be found, e.g., in [9].

A fragment of the retyping relation presented below illustrates a correspondence between the types in the example scenario:

$$\{\mathsf{BookingInfo} \leftrightarrow \mathsf{Order},\ \mathsf{Client} \leftrightarrow \mathsf{Customer},$$
$$\mathsf{Reservation} \leftrightarrow \mathsf{ReservDoc},\ \mathsf{LicenseInfo} \leftrightarrow \mathsf{Authorization},\ \ldots\}$$

Next, we discuss operation declarations, assuming for the moment that the required and provided signatures share the same types. Structural compatibility between two operation declarations is motivated by the following semantic requirements: On the one hand, the provider needs all required inputs in order to execute its operation. On the other hand, the requestor has to rely on the fact that the provided operation returns all expected outputs.

Syntactic criteria ensuring these semantic requirements are based on the matching of input and output type sequences composing the operation declarations. We say that a sequence $\alpha = \alpha_1, \ldots, \alpha_n$ is a *subsequence* of a sequence $\beta = \beta_1, \ldots, \beta_m$, denoted as $\alpha \preceq \beta$, if there exist integers $i_1 < i_2 < \ldots < i_n$ such that $\alpha_j = \beta_{i_j}$ for all α_j. The sequences are *equal*, denoted as $\alpha = \beta$, if they contain the same elements at the same positions. Different notions of structural compatibility between the required $p : v \to w$ and provided $p' : v' \to w'$

Compatibility	*Inputs*	*Outputs*
Exact	$v = v'$	$w = w'$
Input-preserving	$v = v'$	$w \preceq w'$
Output-preserving	$v \succeq v'$	$w = w'$
Generalized	$v \succeq v'$	$w \preceq w'$

Fig. 10. Structural compatibility of operation declarations.

operation declarations are demonstrated in Fig. 10.

The exact compatibility requires operations to have the same inputs and outputs. The next two variants relax the dependencies between the sequences of input and output parameter types, respectively. The last notion of compatibility provides the most general condition, allowing more inputs in the required operation as well as more outputs in the provided operation. An appropriate notion of structural compatibility has to be chosen based on demands of the business domain and the platform on which requestor and provider systems are implemented.

Example 3.1. Let us consider the structural compatibility of the operations reservHotel and bookHotel shown in Fig. 2 and Fig. 3. The declarations of the operations differ in several ways. While the required operation has the extra input LicenseInfo, the provided operation contains the extra output Fulfillment. However, the input offered by the requestor is sufficient for the provider, and the output proposed by the provider satisfies the requestor's expectations. The described deviations between the declarations are permitted under the generalized notion of structural compatibility. △

At this point we are able to formulate a relation over the structural specifications of the entire services, i.e. service signatures. This relation is based on the compatibility criteria developed for the solitary operation declarations.

The intended correspondence between the required and provided signatures is modelled by a signature morphism formally described below.

Definition 3.2 (signature morphism). A *signature morphism* $f = \langle f_{TG}, f_P \rangle : \langle TG, P \rangle \to \langle TG', P' \rangle$ consists of a type graph morphism $f_{TG} : TG \to TG'$ and a mapping of rule names $f_P : P \to P'$ such that for each operation declaration $p : v \to w \in P$, $f_P(p) : v' \to w' \in P'$ with $f^*_{TG}(v) = v'$ and $f^*_{TG}(w) = w'$. △

The definition captures the exact compatibility between the signature declarations. We therefor speak of *exact signature morphisms*. The three remaining variants of structural compatibility can be obtained if the equality between

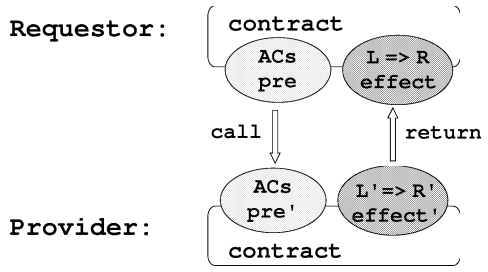

Fig. 11. Compatibility between required and provided operation contracts.

the sequences of input and output parameter types in Def. 3.2 are replaced with the subsequent relations according to Fig. 10, leading to the notions of *input-preserving*, *output-preserving*, and *generalised signature morphisms*.

3.2 Behavioural Compatibility

Behavioural compatibility is verified by relating conditional graph transformation systems describing the required and provided services. To compare behavioural specifications employing different types, their type systems have to be adjusted, i.e. it is necessary to establish a mapping between the required and provided type graphs. This mapping induces a retyping procedure which enables to compare the rules sharing the same type context. We continue our discussion under the assumption that the systems consistently use the same types.

Rule names in a conditional GTS stand for the names of service operations, whose behaviour is specified by contracts taking the form of conditional rules. Before devising syntactic criteria for the behavioural compatibility for a pair of contract rules, we have to understand the semantic requirements.

Fig. 11 shows the invocation of the provided operation by the requestor as well as the provider's reply, together with the relevant preconditions and effects constituting the contracts. The interaction consists of the following steps:

(1) Requestor is willing to submit input data specified in *pre* to issue a call of the provided operation.
(2) Provider assumes that the submitted input satisfies the invocation requirements described in *pre'* and calls its operation.
(3) Provider executes its operation and guarantees the effect described in *effect'*.
(4) Requestor assumes that the provided effect fulfills assumptions specified in *effect* and obtains the result of the operation call.

To make sure that the service implemented by the requestor system with the help of provider works as expected, we have to verify that the assumptions

given in 2 and 4 indeed hold. This would be the case if *pre* implies *pre'*, and *effect'* implies *effect*.

Both semantic requirements will be checked through syntactic criteria on the pair of the required and provided rules $p : s$ if $A(s)$ and $p' : s'$ if $A(s')$, respectively.

The precondition of a rule restricts its applicability. Thus, the first implication is guaranteed if the applicability of the required rule entails that of the provided rule. This is the case if $\hat{L}'_P \subseteq \hat{L}_P$ and $\hat{L}_N \subseteq \hat{L}'_N$. The effect of a rule describes the state changes induced by a transformation step or transition via this rule. Thus, the second implication is ensured if the effect of the required rule is included (and possibly extended) in the provided rule. This is the case if $L \subseteq L', R \subseteq R'$ (left- and right-hand sides are extended), $del(p) \subseteq del(p')$ (more is deleted by p'), and $add(p) \subseteq add(p')$ (more is added by p').

Example 3.3. Let us examine the behavioural compatibility of the operations bookHotel and reservHotel whose contracts are depicted in Fig. 8. To check whether the required operation meets invocation requirements of the provider, we compare the precondition parts of the rules. The invocation requirements of the provided operation are stated in the positive and negative precondition patterns $\hat{L}'_P$ and $\hat{L}'_N$. While the former is enriched in the requestor rule by the object l:LicenseInfo and the edge between this object and the object c:Client, the latter is identical (modulo renaming) to the negative precondition pattern $\hat{L}_N$. That means the applicability of the required rule is extended in the provided rule.

Then, we match the effect parts of the rules, because the benefit obtained via applying the provided operation should satisfy the expectations of the client. The object f:Fulfillment and the edge from this object to the object rd:ResrvDoc are created in the system state after the execution of the operation reservHotel. These elements are not present in the right-hand side of the requestor rule bookHotel, because it is regarded as sufficient to obtain only a confirmation in the form of a reservation tag. Nevertheless, the provided effect extending the required one fits the client requirements. △

Due to the substitution principle declared in the beginning of this section, the abstract rules of the requestor system $\mathcal{C}$ are expected to be replaced by the concrete rules of the provider system $\mathcal{C}'$. Semantically, this means to substitute transformations in $\mathcal{C}'$ for transitions in $\mathcal{C}$, while preserving the applicability of all substituted rules.

Definition 3.4 (substitutability). Given conditional graph transformation systems $\mathcal{C} = \langle TG, P, \pi \rangle$ and $\mathcal{C}' = \langle TG', P', \pi' \rangle$, we say that $\mathcal{C}'$ is substitutable for $\mathcal{C}$ if there exists a functor $F : \mathbf{Trf}(\mathcal{C}') \to \mathbf{Trs}(C)$ such that for all graphs $G' \in |\mathbf{Trf}(\mathcal{C}')|$ and for all transition sequences $\rho : F(G') \to \_ \in \mathbf{Trs}(\mathcal{C})$ there

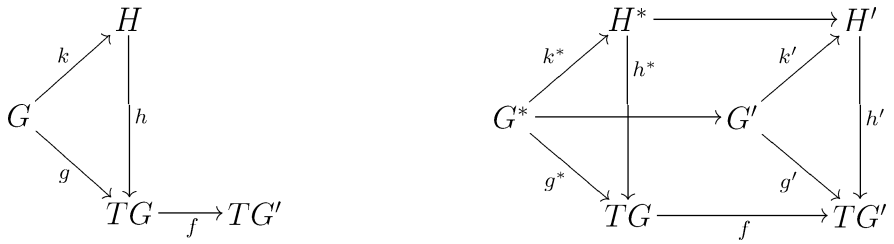

Fig. 12. Forward (left) and backward (right) retyping functors.

exists a transformation sequence $\rho' : G' \to \_ \in \mathbf{Trf}(\mathcal{C}')$ with $F(\rho') = \rho$. △

The operator "_" in the definition denotes the fact that the targets of ρ and ρ' are left unspecified. Functor F translating states of $\mathcal{C}'$ into states of $\mathcal{C}$ can be realised by the retyping induced by a morphism between the two type graphs of the system [18].

Definition 3.5 (retyping). A graph morphism $f_{TG} : TG \to TG'$ induces a *forward retyping functor* $f^{>}_{TG} : \mathbf{Graph}_{TG} \to \mathbf{Graph}_{TG'}$, $f^{>}(g) = f \circ g$ and $f^{>}(k : g \to h) = k$ by composition as shown in Fig. 12 on the left, as well as a *backward retyping functor* $f^{<}_{TG} : \mathbf{Graph}_{TG'} \to \mathbf{Graph}_{TG}$, $f^{<}(g') = g^*$ and $f^{<}(k' : g' \to h') = k^* : g^* \to h^*$ by pullbacks and mediating morphisms as shown in Fig. 12 on the right. △

Now we are ready to formally define behavioural compatibility.

Definition 3.6 (substitution morphism). Given conditional typed graph transformation systems $\mathcal{C} = \langle TG, P, \pi \rangle$ and $C' = \langle TG', P', \pi' \rangle$, a *substitution morphism* $f^{sub} = \langle f_{TG}, f_P \rangle$ is given by a type graph morphism $f_{TG} : TG \to TG'$ and a mapping of rule names $f_P : P \to P'$ such that for each $\pi(p) = s$ if $A(s)$ and $\pi'(p') = s'$ if $A(s')$ where $p \in P$ and $p' = f_P(p) \in P'$

(1) there exist TG-typed graph morphisms $e_L : L \to f^{<}_{TG}(L')$, $e_K : K \to f^{<}_{TG}(K')$, and $e_R : R \to f^{<}_{TG}(R')$ forming a faithful transition (cf. Fig. 13 on the right), and
(2) applicability of $\pi(p)$ implies that of $\pi'(p')$ in $\mathbf{Graph}_{TG'}$, i.e.
   (a) for each $f^{>}_{TG}(L \xrightarrow{\hat{l}} \hat{L}) \in f^{>}_{TG}(AP(s))$ there exist $L' \xrightarrow{\hat{l}'} \hat{L}' \in AP(s')$ and a graph morphism $\hat{h}_P : \hat{L}' \to f^{>}_{TG}(\hat{L})$ such that the corresponding square in Fig. 13 on the left commutes;
   (b) for each $L' \xrightarrow{\hat{k}'} \hat{L}' \in AN(s')$ there exist $f^{>}_{TG}(L \xrightarrow{\hat{k}} \hat{L}) \in f^{>}_{TG}(AN(s))$ and a graph morphism $\hat{h}_N : f^{>}_{TG}(\hat{L}) \to \hat{L}'$ such that the corresponding square in Fig. 13 on the left commutes.

△

Notice that the effect parts of the conditional rules are matched in the type context of $\mathcal{C}$ in contrast with the precondition parts compared in the type

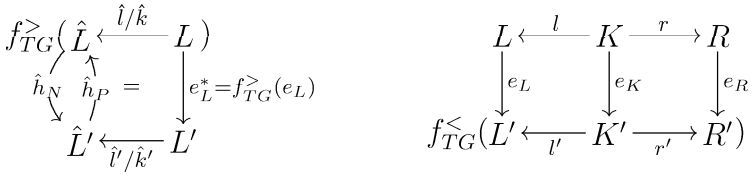

Fig. 13. Substitution morphism of conditional rules (the functors $f^{>}_{TG}$ and $f^{<}_{TG}$ are applied to the entire constraint of p in the left part of the figure and to the entire bottom span in the right part of the figure, respectively).

context of $\mathcal{C}'$. The precondition part of the requestor's rule is forwardly retyped for the matching, because the input data specified by this precondition will be interpreted in the type system of the provider executing the operation. Analogously, the effect part of the provider's rule is backwardly retyped, since the result of the operation will be interpreted and used in the requestor type system.

While the semantic requirements on the structural compatibility are directly reflected by the signature morphism, the correspondence between the semantic requirements on the behavioural compatibility and the substitution morphism is not so obvious. The justifications for the definition of the substitution morphism are presented in the following theorem.

Theorem 3.7. *The substitution morphism $f^{sub} = \langle f_{TG}, f_P \rangle$ satisfies the semantic requirements of Def. 3.4.*

Proof. It is necessary to show that Def. 3.6 implies Def. 3.4. The existence of a functor between two categories of sequences requires that each individual step in $\mathcal{C}'$ is mapped to a sequence in $\mathcal{C}$. By induction, this mapping is extended to sequences in $\mathcal{C}$. However, we will deal with the simpler case where a transformation step in $\mathcal{C}'$ is actually mapped to a single transition in $\mathcal{C}$.

One should demonstrate, first of all, that transformation steps via $\mathcal{C}'$ rule can be considered as transitions via the corresponding $\mathcal{C}$ rule. Secondly, the applicability of this $\mathcal{C}$ rule has to imply the applicability of the $\mathcal{C}'$ rule under the construction of the transformations and transitions associated by the functor F.

Assume two conditional rules $p : s \text{ if } A(s)$ and $p' : s' \text{ if } A(s')$ where $p \in P$ and $p' = f_P(p) \in P'$.

(1) Let us apply backwardly retyped p' to the graph $f^{<}_{TG}(G)$ at $f^{<}_{TG}(d_{L'})$ and construct a transformation step with the underlying span $f^{<}_{TG}(G \xleftarrow{g} D \xrightarrow{h} H)$ as depicted in Fig. 14 on the right. Since l' and r' are injective, this transformation step is also a (faithful) transition. By assumption, for each pair of rules $p : s \text{ if } A(s)$ and $f_P(p) : s' \text{ if } A(s')$ there exists a TG-typed graph morphisms $e_L : L \to f^{<}_{TG}(L')$, $e_K : K \to f^{<}_{TG}(K')$, and

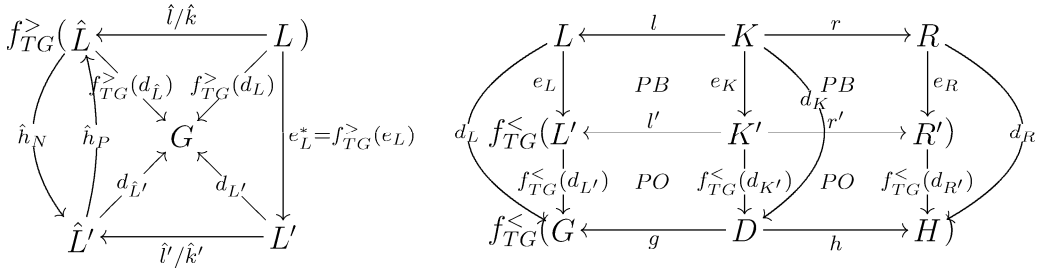

Fig. 14. Substitution morphism and semantic requirements (the functors $f^{>}_{TG}$ and $f^{<}_{TG}$ are applied to the entire application constraint of p in the left part of the figure and to the entire bottom span in the right part of the figure, respectively).

$e_R : R \to f^{<}_{TG}(R')$ forming a faithful transition (cf. Fig. 14 on the right). Now, both transitions can be vertically composed using the composition of the underlying pullback squares. The faithfulness of the composed transition via the rule of $\mathcal{C}$ follows from the preservation of the identification condition under the composition of pullback squares.

(2) It is left to show that if $f^{>}_{TG}(d_L)$ satisfies the application condition of forwardly retyped p, then $d_{L'}$ satisfies the application condition of p'. This induces two problems:
   (a) $d_{L'}$ (cf. Fig. 14 on the left) must *P-satisfy* all positive constraints of p'. Since $\hat{h}_P$ exists by assumption (Def. 3.6.(2a)), $d_{\hat{L}'}$ can be constructed by $f^{>}_{TG}(d_{\hat{L}}) \circ \hat{h}_P$. It is not difficult to see that $d_{\hat{L}'} \circ \hat{l}' = d_{L'}$.
   (b) $d_{L'}$ (cf. Fig. 14 on the left) must *N-satisfy* all negative constraints of p', i.e., there does not exist $d_{\hat{L}'} : \hat{L}' \to G$ such that $d_{\hat{L}'} \circ \hat{k}' = d_{L'}$. Assume to the contrary the existence of $d_{\hat{L}'}$. Since $\hat{h}_N$ exists by assumption (Def. 3.6.(2b)), we can construct $f^{>}_{TG}(d_{\hat{L}}) = d_{\hat{L}'} \circ \hat{h}_N$ with $f^{>}_{TG}(d_{\hat{L}}) \circ \hat{k} = f^{>}_{TG}(d_L)$ which is a contradiction.

   Thus, $d_{L'}$ satisfies the application condition of p'.

Combining the two parts of the proof, we obtain that the functor specified in Def. 3.4 can indeed be constructed. □

3.3 Integral Compatibility

In the previous subsections we have established the relations which model structural and behavioural kinds of compatibility between the interface specifications of the required and provided services. The structural compatibility guarantees the appropriate relationship between input and output parameter types of the associated operations, which may have, however, completely different meanings. The behavioural compatibility, in turn, relates semantically compatible operations, which may be structurally inconsistent.

At this point, we discuss an integral compatibility combining the features of

the structural and behavioural ones. The integral compatibility is represented by a *parameterised substitution morphism* formally defined below. The parameterised substitution morphism associates the required and provided service specifications in the form of conditional parameterised graph transformation systems.

Definition 3.8 (parameterised substitution morphism). Given conditional parameterised graph transformation systems $\mathcal{CP} = \langle TG, P, CP \rangle$ and $\mathcal{CP}' = \langle TG', P', CP' \rangle$. A *parameterised substitution morphism* $f^{sub+} = \langle f_{TG}, f_P \rangle$ is a signature morphism given by a type graph morphism $f_{TG} : TG \to TG'$ and a mapping of rule names $f_P : P \to P'$, where for all $p \in P$ and $p' = f_P(p) \in P'$ the conditional spans of rules $cp = p(x, y) : s \text{ if } A(s) \in CP$ and $cp' = p'(x', y') : s' \text{ if } A(s') \in CP'$ satisfy the requirements of Def. 3.6 $\triangle$

The newly introduced notion of integral compatibility is illustrated by the following example.

Example 3.9. Let us examine the integral compatibility of the operations bookHotel and reservHotel which are specified by conditional parameterised rules in Fig. 8. First of all, one should check the structural compatibility between the parameter type declarations augmenting the rule names. In Example 3.1, we have shown that the declarations of these operations satisfy the requirements imposed by the generalized notion of structural compatibility.

Then, we check the behavioral compatibility which amounts to match the conditional rules associated with the rule names bookHotel and reservHotel. According to Example 3.3, under the appropriate renaming the applicability and effect of the required rule are extended in the provided rule. This guarantees the behavioral compatibility between the considered operations. Combining the outputs of structural and behavioral compatibility tests, we can conclude that the operations bookHotel and reservHotel are indeed integrally compatible. $\triangle$

Two final sections discuss approaches related to our work and summarise the main results.

4 Related Work

The problem of discovering a component or service satisfying specific requirements is not a new one. A significant amount of work has been done in the area of Component-Based Software Engineering (CBSE) to increase the reliability and maintainability of software through reuse. Central here is the development of the techniques for reasoning about component descriptions and component

matching. These techniques differ in the constituents involved in the matching procedure (e.g., operation signatures, behavioural specifications) and the way these constituents are specified (e.g., logic formulas, algebraic specification languages).

One of the most elaborate approaches, along with a thorough overview of related work, is presented by Zaremski and Wing in [19] and [14], who have developed matching procedures for two levels of descriptions (signatures and specifications) and two levels of granularity (functions and modules). Structural and behavioural information about components is given using the algebraic specification language Larch/ML.

A pre/post-condition style of specification, like in [14], is also utilised by other authors. For example, in the work of Perry [20] operations are specified with pre- and post-conditions in first order logic. Order-sorted predicate logic (OSPL) is employed by Jeng and Cheng in [21] for component and function specifications. Basically, two features differentiate our approach from the works described above. The first one is the operational interpretation of graph transformation rules. Second, we have proposed a visual, model-based approach which provides better usability, because it can be more easily integrated into the standard model-driven techniques for software development.

Bracciali et al. in [22] propose a formal methodology for adapting components with mismatching interaction behaviour. An interface specification in this approach consists of two parts: a signature definition describing the functionalities offered and required by a component, and a behaviour specification describing the interaction protocol followed by a component. While signatures are expressed in the style similar to our work, behaviour specifications are expressed by using a subset of π-calculus. Interaction patterns reflected by such specifications can hardly be used to identify the semantics of operations declared in the interfaces, they rather provide add-ons to the operation contracts proposed in our work.

Matching required and provided interfaces is also an issue present in modularisation approaches for algebraic specification languages and typed graph transformation systems (GTS), in particular for the composition of modules. An algebraic module specification MOD in [8] consists of four parts called import IMP, export EXP, parameter PAR, and body BOD. All components are given by algebraic specifications, which are combined through specification morphisms. GTS-modules in [18] are composed of three GTS, IMP, EXP, and BOD, the only difference being the absence of a parameter part. IMP and BOD are related by a simple inclusion morphism, whereas EXP and BOD are connected by a refinement morphism, allowing a temporal or spatial decomposition of rules.

Composition of modules MOD_1 and MOD_2 is based on the morphisms connecting the import interface of MOD_1 with the export interface of MOD_2. Relating required (import) and provided (export) services, this morphism has a similar role to the substitution morphism in this paper. A detailed comparison with [8] is hampered by the conceptual distance between the two formalisms, i.e., graph transformation and algebraic specifications. While the interconnection mechanism in [18] allows an extension and renaming of types, a built-in feature of the substitution morphism, the relation between rules is more general in our case. We have established the most general notion allowing the entailment of applicability from import to export as well as the entailment of effects in the opposite direction.

Finally, let us mention several approaches in the Semantic Web context. Paolucci et al. in [23] propose a solution for automation of Web service discovery based on DAML-S, a DAML-based language for service description. While required and provided service descriptions contain specifications of preconditions and effects, the matching procedure in [23] merely compares input and output parameters of services. Such a kind of matching can be considered as an extended variant of the structural compatibility. Sivashanmugan et al. in [24] extend WSDL using DAML+OIL ontologies to support semantics-based discovery of Web Services. The authors emphasise the importance of matchmaking not only for input and output parameters, but also for functional specifications of operations. Since the work contains only conceptual descriptions of the matching procedure, we can not provide a more formal analysis.

Hausmann et al. [4] use graph transformation rules defined over a domain ontology to represent service specifications and introduce a matching relation between them. The strength of this work is the implementation of the matching procedure in a prototypical tool chain. Informally, the ideas introduced in [4] are similar to those of this paper, but there are technical differences. While the matching procedure has been defined in a set-theoretic notation, the authors of [4] did not provide a formal operational semantics along with semantic requirements towards the desired procedure. As a consequence, the correctness of the proposed formalism has been justified only by means of examples. Besides, the lack of application conditions limits the expressiveness of contracts to positive statements. As already mentioned in the introduction, the additional flexibility gained through the matching of contracts over different data type models is an original contribution of this paper, not only with respect to [4], but also to our own work published in [2,3].

Another approach is presented by Pahl in [13]. He proposed to use description logic for service specification and introduced a contravariant inference rule capturing service matching. This approach is closely related to ours because of the pre/post style of service specification and the contravariant character of

the matching. But, as it was correctly stated by the author, the expressiveness of description logic has negative implications for the complexity of reasoning. Unlike our approach, the service matching in [13] has a problem with decidability that can be guaranteed only under certain restrictions (the set of predicates must be close under negation). Although, the sub-graph problem, in general, is not solvable in polynomial time, there exist a number of heuristic solutions which make it appear realistic [25].

5 Conclusion and Future Work

In this paper we have established a formal background for an approach to service specification and matching based on conditional graph transformation. While the structural compatibility between the interface specifications of the required and provided services was modelled by a signature morphism, the behavioural compatibility was specified by a substitution morphism over the graph transformation systems containing operation contracts in the form of conditional rules. We have used a loose interpretation of rules via DPB graph transitions in order to obtain an operational understanding of contracts as well as appropriate semantic requirements guaranteeing the behavioural compatibility. It has been demonstrated that substitution morphisms meet these semantic requirements. The two aspects of compatibility have been absorbed by the integral compatibility being ensured via a *parameterised substitution morphism* relating graph transformation systems with conditional parameterised rules.

Several questions remain for future work. First, the formal presentation should be extended towards typed graphs with attributes and subtyping [26]. Second, the compatibility relation could be improved to allow the matching of one requestor rule against a spatial/temporal composition of several provider rules [18].

Third, the practical application of the concepts discussed in this paper requires a mapping to the Web service platform, consisting of XML-based standards like SOAP and WSDL. The first part of this mapping has to relate the type systems of both levels, i.e. a type graph of GTS and an XML-schema of a WSDL specification. The second part should associate operation signatures given by UML interfaces with the corresponding specifications of operations in a WSDL document. The last part should provide the mapping of contracts into an adequate XML-representation. This should be integrated with WSDL and should support the implementation of the corresponding matching procedures. While there are isolated examples for all three mappings in the literature, their integration yet remains an open issue.

References

[1] D. Fensel, C. Bussler, The web service modeling framework WSFM, Electronic Commerce Research and Applications 1 (2) (2002) 113–137.

[2] A. Cherchago, R. Heckel, Specification matching of web services using conditional graph transformation rules, in: H. Ehrig, G. Engels, F. Parisi-Presicce, G. Rozenberg (Eds.), Proc. 2nd Int. Conference on Graph Transformation (ICGT'04), Rome, Italy, Vol. 3256 of LNCS, Springer-Verlag, 2004, pp. 304–318.

[3] R. Heckel, A. Cherchago, M. Lohmann, A formal approach to service specification and matching based on graph transformation, in: Proc. First International Workshop on Web Services and Formal Methods (WS-FM 2004) February 23-24, 2004, Pisa, Italy, 2004.

[4] J.-H. Hausmann, R. Heckel, M. Lohmann, Model-based discovery of web services, in: Proc. 2004 IEEE International Conference on Web Services (ICWS 2004) July 6-9, 2004, San Diego, California, USA, 2004.

[5] H. Ehrig, M. Pfender, H. Schneider, Graph grammars: an algebraic approach, in: 14th Annual IEEE Symposium on Switching and Automata Theory, IEEE, 1973, pp. 167–180.

[6] R. Heckel, H. Ehrig, U. Wolter, A. Corradini, Double-pullback transitions and coalgebraic loose semantics for graph transformation systems, Applied Categorical Structures 9 (1), see also TR 97-07 at `http://www.cs.tu-berlin.de/cs/ifb/TechnBerichteListe.html`.

[7] A. Habel, R. Heckel, G. Taentzer, Graph grammars with negative application conditions, Fundamenta Informaticae 26 (3,4) (1996) 287 – 313.

[8] H. Ehrig, B. Mahr, Fundamentals of algebraic specification 2: module specifications and constraints, Springer-Verlag, 1990.

[9] The OpenTravel™Alliance, The OpenTravel™Alliance Specification version 2005A, `http://www.opentravel.org/2005a.cfm` (June 2005).

[10] G. Alonso, F. Casati, H. Kuno, V. Machiraju, Web Services: Concepts, Architectures and Applications, Springer-Verlag, 2004.

[11] B. Meyer, Object-oriented Software Construction: Second Edition, Prentice Hall, 1997.

[12] D. Fensel, Ontologies: Silver Bullet for Knowledge Management and Electronic Commerce, Springer-Verlag, 2001.

[13] C. Pahl, An ontology for software component matching, in: M. Pezze (Ed.), Fundamental approaches to software engineering: 6th international conference, FASE 2003, Vol. 2621 of LNCS, Springer, 2003, pp. 6–21.

[14] A. Zaremski, J. Wing, Specification matching of software components, in: Proc. SIGSOFT95 Third ACM SIGSOFT Symposium on the Foundations of Software Engineering, Vol. 20(4) of ACM SIGSOFT Software Engineering Notes, 1995, pp. 6–17, also CMU-CS-95-127, March, 1995.

[15] A. Corradini, U. Montanari, F. Rossi, Graph processes, Fundamenta Informaticae 26 (3,4) (1996) 241–266.

[16] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, M. Löwe, Algebraic approaches to graph transformation, Part I: Basic concepts and double pushout approach, in: G. Rozenberg (Ed.), Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations, World Scientific, 1997, pp. 163–245, preprint available as Tech. Rep. 96/17, Univ. of Pisa, `http://www.di.unipi.it/TR/TRengl.html`.

[17] H. Ehrig, Introduction to the algebraic theory of graph grammars, in: V. Claus, H. Ehrig, G. Rozenberg (Eds.), Proc. 1st Graph Grammar Workshop, Vol. 73 of LNCS, Springer Verlag, 1979, pp. 1–69.

[18] M. Grosse-Rhode, F. Parisi-Presicce, M. Simeoni, Refinements and modules for typed graph transformation systems, in: J. Fiadeiro (Ed.), Proc. Workshop on Algebraic Development Techniques (WADT'98), at ETAPS'98, Lisbon, April 1998, Vol. 1589 of LNCS, Springer-Verlag, 1999, pp. 138–151.

[19] A. Zaremski, J. Wing, Signature matching: a tool for using software libraries, ACM Transactions on Software Engineering and Methodology (TOSEM) 4 (2) (1995) 146 – 170.

[20] D. E. Perry, S. S. Popovich, Inquire: Predicate-based use and reuse, in: Proc. of the 8th Knowledge-Based Software Engineering Conference, 1993, pp. 144–151.

[21] J.-J. Jeng, B. H. C. Cheng, Specification matching for software reuse: A foundation, in: Proc. of the ACM SIGSOFT Symposium on Software Reusability (SSR'95), 1995, pp. 97–105.

[22] A. Bracciali, A. Brogi, C. Canal, A formal approach to component adaptation, Journal of Systems and Software 74 (1) (2005) 45–54.

[23] M. Paolucci, T. Kawmura, T. Payne, K. Sycara, Semantic matching of web services capabilities, in: Proc. First International Semantic Web Conference, 2002.

[24] K. Sivashanmugam, K. Verma, A. Sheth, J. Miller, Adding semantics to web services standards, in: Proc. First International Conference on Web Services (ICWS03), 2003.

[25] A. Zündorf, Eine Entwicklungsumgebung für programmierte graphersetzungssysteme (PROGRES), Ph.D. thesis, RWTH Aachen (1995).

[26] G. Taentzer, A. Rensink, Ensuring structural constraints in graph-based models with type inheritance., in: M. Cerioli (Ed.), FASE'05, Vol. 3442 of Lecture Notes in Computer Science, Springer, 2005, pp. 64–79.