Evolving Services from a Contractual Perspective*

Vasilios Andrikopoulos[1], Salima Benbernou[2], and Mike P. Papazoglou[1]

[1] ERISS, Tilburg University, Netherlands
[2] LIRIS, Université de Lyon 1, France
{v.andrikopoulos,mikep}@uvt.nl, sbenbern@liris.univ-lyon1.fr

Abstract. In an environment of constant change, driven by competition and innovation, a service can rarely remain stable - especially when it depends on other services to fulfill its functionality. However, uncontrolled changes can easily break the existing relationships between a service and its environment (its customers and providers). In this paper we present an approach that allows for the controlled evolution of a service by leveraging the loosely-coupled nature of the SOA paradigm. More specifically, we formalize the notion of contracts between interacting services that enable their independent evolution and we investigate under which criteria can changes to a contract-bound service, or even to the contract itself, be transparent to the environment of the service.

Keywords: service evolution, service contracts, compatibility, contract invariance, contract evolution.

1 Introduction

A number of serious challenges like mergers and acquisitions, outsourcing possibilities, rapid growth, regulatory compliance needs and intense competitive pressures require changes at the enterprise level and lead to a continuous business process redesign and improvement effort. Service changes that are required by this effort however must be applied in a controlled fashion so as to minimize inconsistencies and disruptions by guaranteeing seamless interoperation of business processes that may cross enterprise boundaries.

In general, we can classify service changes depending on their direct and side effects [1] in *shallow*, where the change effects are localized to the service or are strictly restricted to the clients of that service, and *deep*, that are cascading types of changes which extend beyond the clients of a service, and possibly to its entire value-chain, i.e., to clients of the service clients such as outsourcers or suppliers. Shallow changes characterize both singular services and business processes and require a structured approach and robust versioning strategy to

* The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube).

support multiple versions of services and business protocols. Deep changes on the other hand are more intricate and require the assistance of a *change-oriented service life cycle* where the objective is to allow services to predict and respond appropriately to changes as they occur [1]. Due to the complexity and scope of deep changes this paper discusses only shallow changes and more specifically changes to the Structural layer elements of the Service Specification Reference Model introduced in [2], i.e., to the message content, operations, interfaces, and message exchange patterns (MEPs), roughly corresponding to WSDL artifacts.

The setting discussed has a number of similarities with the fields of *evolution transparency* and *interoperability preservation* that have been discussed in different forms in [3] and [4] (among others), and essentially boil down to preventing incompatibility between interoperating (interacting) services. These works though depend mainly on adaptation mechanisms to maintain interoperability, and adaptation approaches are by definition *a posteriori* interventions focusing on incompatibility identification and resolution by modification of a service. In that sense, the adaptation process can not discern between shallow and deep changes and is unable to prevent the propagation of changes throughout the value chain, since the modification of a service may have unforeseen consequences to the parties that interact with it. For that reason we are focusing on identifying under which conditions changes to a service are shallow and discuss an *a priori* approach that aims to prevent or at least predict and confine the necessity for adaptation.

The goal of this work is therefore to allow the *independent* evolution of loosely coupled interacting parties in a *transparent* manner so as to preserve their interoperability. In this context, the parties involved in an interaction can either be services, or services and client (service-based) applications. We only consider bilateral interactions, and for each such interaction we distinguish two roles: that of the *producer* and that of the *consumer*. It must be kept under consideration that the role of a service, unlike that of an application that always acts as a consumer, can vary depending on the interaction. An aggregate service for example plays both roles: that of the producer for its clients, and that of the consumer when it interacts with the aggregated services to compose a result. To achieve meaningful interoperability in this context, service clients and providers must come to a mutual agreement, a *contract* of sorts between them [5]. A contract of this type formalizes the details of a service in a way that meets the mutual understanding and expectations of both service provider and service client. Building around this idea, we are presenting mechanisms to effectively deal with the evolution of the structural aspect of both parties, while preserving interoperability despite the changes that may affect them. After we lay down this foundation we discuss the evolution of interactions and contracts themselves.

The rest of the paper is organized as follows: section 2 presents a notation for service description that leverages the decoupling of service providers and clients through the introduction of the contract construct (section 3). Section 4 shows how the introduced notions can be used to control the evolution of the interacting parties while maintaining a high degree of flexibility. Section 5 will briefly present related works, and section 6 discusses conclusions and future work. To facilitate the conversation, we are using the simple service described below as a point of reference:

Example 1 (Running Example). Let's assume the case of a very simple inventory service that checks for the availability of an item and responds that the purchase order can be fulfilled or issues a fault stating that the order cannot be completed. The WSDL file of this service is shown in listing 1.

```
...
  <types>
    <xsd:schema targetNamespace="http://e-grocery.com/InventoryService">
      <xsd:complexType name="inventoryItem">
        <xsd:sequence>
          <xsd:element name="orderID" type="xsd:string"/>
          <xsd:element name="itemID" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:schema>
  </types>
  <message name="InventoryRequest">
    <part name="inventoryItem" type="tns:inventoryItem"/>
  </message>
  <message name="InventoryConfirmation">
    <part name="confirmationMessage" type="xsd:string"/>
  </message>
  <message name="InventoryFault">
    <part name="faultMessage" type="xsd:string"/>
  </message>
  <portType name="InventoryServicePortType">
    <operation name="checkInventory">
      <input name="item" message="tns:InventoryRequest"/>
      <output name="confirmation" message="tns:InventoryConfirmation"/>
      <fault name="fault" message="tns:InventoryFault"/>
     </operation>
  </portType>
...
```

Listing 1. Inventory Service WSDL specification

2 Service Specifications

The WSDL description of the inventory service in listing 1 is far from complete in describing the structural aspect of the service. In specific, apart from providing an unambiguous schema for the service interfaces (the *signature* of the service) to be used by its clients, it lacks completely in providing *a)* any information on the services used by the service itself to fulfill its functionality (if any), and *b)* the means to connect the information *required* and *provided* by its signatures with that of the signatures of the other services it is using. It is therefore not suitable for describing the interaction of the service with its environment and has to be replaced by a declarative specification that fulfills this role. [2] provides a more exhaustive discussion on the structure and content of such a service specification scheme. For the purposes of this work, we will only define the following constructs:

Definition 1 (Element). *An* element *e of a service s is defined as a tuple $(a_1, a_2, \ldots, a_n)$, the set of attributes that characterize the element. a_i is either an atomic attribute or another element e_i of the service.*

For example, `InventoryRequest`, `checkInventory`, and the rest of the WSDL constructs in listing 1 can be represented as elements $a_1 =$`(inventoryItem)`, $a_2 =$`(item, confirmation, fault)`, etc.

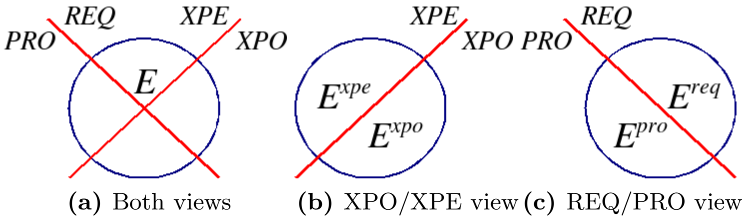
Definition 2 (Type extension). *The* specification *E of a service is defined by the set $E = \{e_i, i \geq 1\}$ of its elements. We associate to E the reflexive and transitive relation* type extension *$\leq$ on elements $(E, \leq)$ defined as: $e \leq e' \Leftrightarrow \{a_1, \ldots, a_n\} \subseteq \{a'_1, \ldots, a'_m\}, m \geq n \wedge a_i \leq a'_j, 1 \leq i \leq n, 1 \leq j \leq m$.*

If for example $\exists a'_1 =$ `(orderID,itemID,comment)` (as in listing 1 in section 3.1) then: `(orderID,itemID)` $\subseteq$ `(orderID,itemID,comment)` $\Rightarrow a_1 \leq a'_1$, i.e., the new `(inventoryItem)` is a type extension of the old one.

As discussed in the previous section, the approach discussed by this work assumes that *a)* both producer and consumer are in the general case services, and therefore use the same notation to describe their specifications, and *b)* a producer in one interaction can also act as the consumer for another interaction. This latter interaction may or may not be related to the producer's function in the former. Beyond this generic case, the same paradigm can also be applied to "simpler" cases: autonomous services that implement all of their offered functionalities without using other services act only as producers. Non-service clients (e.g., GUI-supported applications) can be perceived as special cases of exclusive consumers.

We define two orthogonal *views* on E (see figure 1a): the expositions/expectations view and the required/provided view:

These views provide us with reference points in disambiguating the roles and functions of elements in a service specification like listing 1. More specifically:



(a) Both views **(b)** XPO/XPE view **(c)** REQ/PRO view

Fig. 1. Views on the service specification

2.1 Exposition/Expectation View

This view (figure 1b) classifies the elements within a service specification with respect to whether they are offered as an interface to the environment or they are “imported” into the service specification, referring to interface elements of other services. In the former case, the service acts as a producer; in the latter as a consumer. Elements of a service specification can therefore fall into one of the following categories:

– Exposition E^{xpo}: the set of elements that describe the offered functionality of the service.
– Expectation E^{xpe}: the set of elements describing the perceived offering of functionality to the service by other services.

The WSDL file of the inventory service for example in listing 1 contains the information on how to access the elements that constitute the inventory service and what information is exchanged while accessing it. From the perspective of the producer of the service, this file specifies what the producer will offer to the service customers: if the `checkInventory` operation is invoked using the `InventoryServicePortType` and the message payload defined, the generated result or fault message will be a simple string. The elements of the file are in that sense in the exposition subset of the service producer specification.

On the other hand, when a consumer of this service builds and/or uses an application that incorporates an invocation of this service, the consumer refers to what it *perceives* to be a set of elements that allow it to access the service. To put it simply, the client is built on the premise of a particular specification of the provided interface, being bound for example to the service of listing 1. These elements are therefore contained in the expectation subset of the consumer specification. What becomes apparent from this is that the same elements can either be expositions or expectations; it only depends on the adopted viewpoint.

Ideally, this perceived specification and the actual specification of the provided service are the same - and that is so far the fundamental assumption in service interactions. But changes to either side, as we will discuss in the following sections, could lead to inconsistencies - in other terms incompatibilities - between those two.

2.2 Required/Provided View

The division enforced by this view (figure 1c) is much more straightforward: it provides the means to cleanly separate input from output in a service specification (irrespective of whether it acts as a producer or a consumer). More specifically:

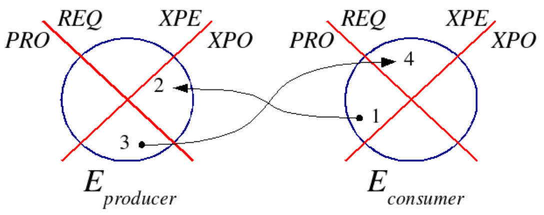
– Required E^{req}: contains the input-type elements of the service specification.
– Provided E^{pro}: contains the output-type elements.

`InventoryRequest` for example is clearly a required element for the producer: it is the input message type for the service. At the same time it is a provided element for the consumer since it has to be provided to the producer in order to use the respective operation. `InventoryConfirmation` and `InventoryFault` are respectively provided elements for the producer - they are produced as output by the service in one way (normal result) or another (fault message) - and required elements for the consumer (input to it).

2.3 Combining the Views

Since the two views are orthogonal, they can be used in conjunction to describe the elements of a service specification: $E^{xpo} \cup E^{xpe} = E^{req} \cup E^{pro} = E$ (figure 1a).

Example 2. Figure 2 shows how an invocation of the inventory service of listing 1 by a Web services client can be described using the classification presented. Due to the request-response messaging pattern of the `checkInventory` operation, the interaction between the service and its client is broken down into two phases: in the first phase, the consumer (client) is using the expectation element *(1)* to invoke the exposition element *(2)* of the producer (service). Since *(1)* is an output for the consumer it belongs to the $E^{pro}_{consumer}$ set, and *(2)* is in the $E^{req}_{producer}$ as the input of the service. The situation is inversed for the second phase, where the producer uses *(3)* to call back *(4)* in the consumer side.



	Element
1	Consumer:`InventoryRequest`
2	Producer:`InventoryRequest`
3	Producer:`InventoryResponse`
4	Consumer:`InventoryResponse`

Fig. 2. Service Interaction

3 Contracts

This section builds on the notation and classification presented in the previous section to discuss the interaction of parties in a loosely-coupled environment and introduce the notion of *contracts* as the means to leverage the decoupling between producer and consumer.

By the term contract we do not refer to the legal documents that describe a binding agreement, but we use the term in the same manner as the (software) contracts in the Eiffel language [6]. The contracts in this context are documents that record the *benefits* expected by each party from their interaction, and the *obligations* that each party is prepared to carry out in order to obtain these promised benefits. In that sense, the contract protects both sides by clearly defining what is the acceptable contribution and result for a task described by the contract. Our approach applies the same paradigm on services specifications, using the different views discussed above to distinguish between those benefits and obligations, depending on the role that the service plays.

In specific, there is an important distinction in the way that the producer and the consumer of a service are perceiving a service specification document: the producer promises to offer the service in the manner specified in it (the expositions set), and the consumer accepts this promise and builds a client for it based on this promise (the expectations set). In most contemporary SOA implementations, by using for example Web services technologies, this fundamental difference is bridged by accepting one perspective, that of the producer, and shifting the consumer side perspective accordingly. But in this case the consumer has to adopt any changes and assumptions that are done by the producer. Failure to comply with the producer means that the consumer is unable to use the offered functionality, which explains why producer updates typically fail on the client side.

In order to amend this situation, we propose to use a construct (the contract) that bridges the two perspectives and allows for mapping from and to it by either party. This contract is nothing more than an intermediary specification, containing a set of commonly agreed elements specified in a party-independent way. By providing a neutral mapping procedure from each party to the contract we minimize the producer/consumer coupling. Furthermore, given a contract, we allow for reasoning by each party *in isolation*, enforcing the separation of concerns and responsibilities in service design and operation. In the following we formally define the contract construct and describe how to formulate a contract between two parties.

3.1 Contract Definition

In principle only a part of the offered service functionalities may be used by a specific client; on the other hand, a client may depend on a number of disparate services in order to achieve its goals. Thus we need a way to identify and isolate the parts of the interacting parties that actually contribute to the interaction. For that purpose we will denote with $P \subseteq E^{xpo}_{producer}$ and $C \subseteq E^{xpe}_{consumer}$ the subsets from the producer and consumer specifications respectively that participate in the interaction.

Following on we define a binding function ϑ that reasons *horizontally* between the elements of parties P and C:

Definition 3 (Service Matching). *A service matching is a binding function defined as* $\vartheta : P \times C \to U, U = P \cup C$ *such that*

$$\vartheta(x,y) = \{z \in U / \begin{cases} x \preceq z \preceq y, x \in P^{req}, y \in C^{pro} \\ y \preceq z \preceq x, x \in P^{pro}, y \in C^{req} \end{cases} \} \tag{1}$$

Example 3. Let's assume that P contains the elements of listing 1 and let's denote by $x \in P^{req}$ the `InventoryItem` element: $x = (a_1, a_2)$, $a_1 =$ `(orderID)` and $a_2 =$ `(itemID)`. A consumer of this service that is bound to listing 1 uses all elements as they are defined in the listing (that is: $P \equiv C$) and therefore

$\exists y \in C^{pro} / y = x \Rightarrow \vartheta(x, y) = z = (a_1, a_2)$. This reasoning holds also for the rest of the elements of P and C and the service matching is in that case trivial.

Now consider the case of another consumer C' that is bound to listing 2 that differs from listing 1 in the definition of `InventoryItem`: $y' = (a_1, a_2, a_3)$, $a_3 =$ `(comment)` to allow for attaching notes to items. By its definition $\vartheta(x, y') = z'$ returns two possible values: $z' = (a_1, a_2)$ or $z' = (a_1, a_2, a_3)$. By selecting the first value (reflecting the assumption that P can *ignore* this extra argument in the requests of C') we observe that the previous service matching between P and C persists for P and C' despite the changes in consumer C. The actual selection of the binding function value during the contract formulation is a matter of policy (see following section for a further discussion on this subject).

```
...
    <xsd:complexType name="inventoryItem">
      <xsd:sequence>
        <xsd:element name="orderID" type="xsd:string"/>
        <xsd:element name="itemID" type="xsd:string"/>
        <xsd:element name="comment" type="xsd:string" minOccurs="0"/>
      </xsd:sequence>
    </xsd:complexType>
...
```

Listing 2. Alternative inventory item definition

Binding function ϑ is acting in the same manner as a *schema matching* function would. Schema matching aims at identifying semantic correspondences between elements of two schemas, e.g., database schemas, ontologies, and XML message formats [7]. It is necessary in many database applications, such as integration of web data sources, data warehouse loading and XML message mapping. In most systems, schema matching is manual or semi-automatic; a time-consuming, tedious, and error-prone process which becomes increasingly impractical with a higher number of schemas and data sources to be dealt with. In our case though, the matching function relies on the type extension relation to automatically identify elements on either party that are semantically related to each other according to their respective schemata.

Based on the matching function ϑ we can define the *Contract R* between two parties as a service mapping:

Definition 4 (Service Mapping). *A Service mapping is a* Contract *R defined by a triplet $< P, C, \Theta >$ between the two parties that is defined as their image under ϑ, i.e., $\Theta = \{\vartheta(x, y) | x \in P, y \in C\}$. The elements that comprise R are called the* clauses *of the contract.*

The mapping therefore consists of the results of the binding function for all possible pairs in the producer/consumer sets and is formulated by reasoning vertically through the parties. The contract that is produced by this mapping

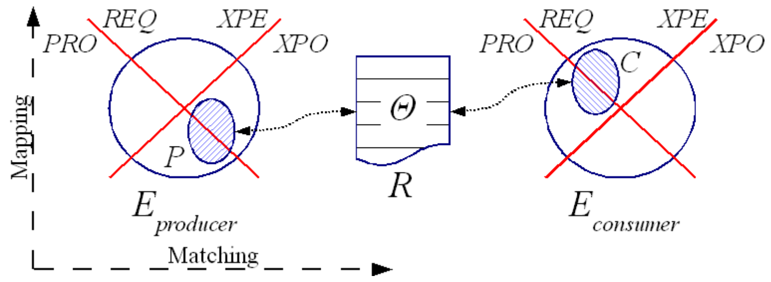


Fig. 3. Producer/Consumer/Contract relation

identifies and represents the mutually agreed specification elements that will be used for the interaction of the parties. Figure 3 demonstrates the relation between P, C, and R graphically.

Example 4. Following the previous example, the service mapping between P and C (consumer using listing 1) would consist of the contract $R =< P, C, \Theta >, P \equiv C \equiv \Theta$.

For the service mapping between P and C' (consumer using listing 2) we are presented with two options: either we opt for $z' = (a_1, a_2)$ and since the rest of the elements remain the same then $\Theta' \equiv \Theta \Rightarrow R' \equiv R$, or in the case of selecting $z' = (a_1, a_2, a_3)$ then a new contract $R' =< P, C', \Theta' >$ has to be formulated.

3.2 Contract Formulation and Management

The definition of contract R between two parties as a service mapping $< P, C, \Theta >$ allows for a straightforward formulation of the contract: given the two parties' specifications P and C, each of which defines the elements through which the interaction is achieved, Θ can be calculated directly by applying the matching function ϑ to them. The issue of contract development therefore shifts in producing P and C from the service provider $E^{xpo}_{producer}$ and client $E^{xpe}_{consumer}$ specifications respectively.

Due to the fact that the service provider is unaware of the internal workings of the service client (represented by the $E^{xpe}_{consumer}$ set) the process of contract formulation is consumer-driven; more specifically, the steps to be followed are:

1. The consumer decides on the functionality offered by the producer that will be used (if more than one is offered).
2. The set of elements from $E^{xpo}_{producer}$ that fulfill this functionality (e.g., the port type and the associated structural elements) are identified.
3. The identified elements are either copied to the (initially empty) $E^{xpe}_{consumer}$ set or the existing $E^{xpe}_{consumer}$ set is used.
4. The image of P and C under ϑ set is calculated. If the resulting set is empty then the image is attempted to be re-calculated using alternative values from ϑ (or cancelled in case all possibilities have been exhausted); otherwise the contract $R =< P, C, \Theta >$ is produced.

5. The consumer submits the formulated contract R to the producer for posterity and begins interaction with producer.

The formulating, storing, and reasoning aspects of the proposed solution can be incorporated in the service governance infrastructure that supports each party. Since ϑ may return one or more possible values, depending on the type extension 'distance' in the element definition between the producer and consumer specification, a minimum level of 'insight' on the consumer side is required in selecting the appropriate elements from the producer and in assigning values to the binding function ϑ:

Conservative selection policies would opt for the values contributed by the consumer to the calculation of ϑ, trying to protect the consumer from possible changes to the producer.

Liberal selection policies on the other hand would pick the values contributed by the producer and allow for the possibility of the consumer evolving in the future.

The type of policy to be followed is therefore largely a design and governance issue and has to be dealt as such. The solution presented assumes that producers and consumers have the means to formulate, exchange, store, and reason on the basis of contracts. In absence of these facilities from one or both parties the interaction between them reverts to the non contract-based modus operandi that, as we have discussed above, can not guarantee interoperability. The exchange of contracts requires the existence of a dedicated mechanism for this purpose that is not part of the service specification.

4 Contract-Controlled Service Evolution

The previous section discussed how to leverage the loose coupling of the producer and the consumer by means of the contract construct. The following section discuss how this design solution enables evolutionary transparency that preserves (under certain conditions) the producer/consumer interoperability.

In the initial 'static' state of two interoperating parties P and C, and after a contract $R =< P, C, \Theta >$ has been formulated and accepted between them, it holds in general that $P \equiv \Theta \equiv C$. For example, when a simple client is using the service described in listing 1 it is safe to assume that due to the granularity of the service, the client will be using the one (and only) functionality provided by it. That in turn means that it will refer to all the elements contained in the WSDL file. Therefore, $P \equiv C$ and by the definition of the contract construct, $P \equiv \Theta \equiv C$, as we have seen in the previous section.

But since either party can, or at least should be able to evolve independently of the other, shifts from this state can occur. When changes for example occur to the producer then it may hold that $P' \not\equiv \Theta \equiv C$, or for the consumer side $P \equiv \Theta \not\equiv C'$, or both. These latter states reflect situations of incompatibility between producer and consumer and they have to be prevented from occurring in order to avoid the occurrence of deep changes in the context of the interacting parties. The introduction of a contract between them allows us to reason about the contribution of each party to the interaction without directly affecting the other party, ensuring that each party is able to evolve *independently* but *transparently*, that is without requiring modifications, to each other.

For that purpose we will distinguish *shallow* changes occurring to a party in two categories: those that respect the *contractual invariance* and those that require *contractual evolution.* Changes to a party that fall in the former category do not affect the existing contract between the parties. Changes in the latter category require modifications to the contract but nevertheless do not require changes to the other party.

4.1 Contract Invariance

Taking advantage of the ability to reason exclusively on one party given an existing contract, without the need for the other party to participate in this reasoning, exemplifies the notion of independence in evolution. In order to show how this is accomplished we will first formally define what it means for a (modified) party specification to respect, or to be *compliant* with a contract:

Definition 5 (Compliance to Contract). *A version of a party, e.g. version P' of producer P, is said to be* compliant *with respect to an existing contract $R =< P, C, \Theta >$ with a consumer C denoted by $P' \vDash_R C$ iff*

$$\forall z \in \Theta / \exists x' \in P', \vartheta(x', y) = z, y \in C \tag{2}$$

Corollary 1. *Consequently, P'* violates *R, and we write $P' \nvDash_R C$, iff $\exists z \in \Theta / \forall x' \in P', \vartheta(x', y) \neq z, y \in C$.*

The definition above allows for a simple algorithm to check for the compliance of a new version of a party in the producer-consumer relationship: as long as there is a mapping produced by ϑ to *all* clauses of the contract from the elements of the new specification, the two versions are equivalent or *compatible* with respect to the contract - or more formally:

Definition 6 (Compatibility w.r.t. existing Contract)

1. *Given a party, e.g. consumer C, then two versions of the other party, P and P', are called* compatible *w.r.t. a contract R denoted by $P \mapsto_R P'$ iff they are both compliant to R: $P \vDash_R C \wedge P' \vDash_R C$.*
2. *Two versions of a party S and S' are called* fully compatible *iff they are compatible for all contracts $R_i, i \geq 1$ that they participate in, either as producers or consumers: $S \mapsto_{R_i} S' \; \forall R_i$.*

Example 5. Consider the modifications applied to the service specification as depicted in listing 3. Let's assume that these changes are applied to P; in that case P' is compatible with P, since they are both compliant to the same contract R. To prove that, we start with the observation that element $x =$(`InventoryConfirmation`) in listing 1 is in the P^{pro} set, and therefore contributes to the second leg of the binding function (1) which means that

$$\exists y \in C^{req}, z \in \Theta / y \leq z \leq x. \tag{3}$$

Let's denote with x' the changed element from listing 4.1. It holds that $x \leq x'$ and in conjunction with (3) we get: $\exists y \in C^{req}, z \in R / y \leq z \leq x'$. Thus, $\vartheta(x', y) = \vartheta(x, y)$, and since the rest of the matchings remain unchanged, by (2) we can deduce that $P' \models_R C$.

If listing 3 though is depicting changes to the consumer side, then by the same reasoning we can easily prove that C and C' are not compatible, since $P \not\models_R C'$.

```
...
<message name="InventoryConfirmation">
  <part name="confirmationMessage" type="xsd:string"/>
  <part name="confirmationDate" type="xsd:date"/>
</message>
...
```

Listing 3. New inventory Service WSDL specification

4.2 Contract Evolution

The previous section discussed the criteria under which changes to one party can leave the contract between them intact, essentially ensuring that these changes are shallow. This does not necessarily mean that all changes that do not respect this criteria are deep. The existing interaction between the parties can be preserved in certain cases despite the necessity to modify the contract due to changes to one or both of the parties involved, defined as *backward* and *forward* compatibility preserving cases:

Definition 7 (Backward Compatibility). *Two contracts $R =< P, \Theta, R >$ and $R' =< P, \Theta', C' >$ are called* backward compatible *and we write $R \mapsto_b R'$ iff $\forall x \in P / \exists z' \in \Theta', \exists y' \in C', z' = \vartheta(x, y')$.*

In that case changes to the consumer side leave the producer unaffected. The (new) consumer will use the producer in the same manner as the old consumer did.

Definition 8 (Forward Compatibility). *Two contracts $R =< P, \Theta, R >$ and $R' =< P', \Theta', C >$ are called* forward compatible *and we write $R \mapsto_f R'$ iff $\forall y \in C / \exists z' \in \Theta', \exists x' \in P', z' = \vartheta(x', y)$.*

Forward compatibility therefore allows for the seamless interoperation of the new producer with the old consumer without the former party to have to be modified in any way.

It must be noted that these definitions following [1] are using the vantage point of the *consumer* to discuss changes: a change to a contract is backwards compatible if it allows the consumer to accept input from older devices (versions of the producer). Similarly, a forwards compatible contract means that the consumer can accept input from newer versions of the producer. Consider for example the discussion in section 3.1 on the possibilities for service matching and mapping: if we choose to create a new contract R' then it can be easily shown that this contract is backward compatible to R and therefore the new consumer C' can still use the old producer P.

Furthermore, by combining the two definitions we can define when two contracts are compatible:

Definition 9 (Contract Compatibility). *Two contracts $R =< P, \Theta, R >$ and $R' =< P', \Theta', C' >$ are called compatible and we write $R \mapsto R'$ iff they are both backward and forward compatible: $R \mapsto_b R' \wedge R \mapsto_f R'$.*

Contrary to the case of contractual invariance, evolution of the contract itself requires of the parties to exchange a new contract and replace the old contract with the new one. This creates an additional communication overhead that nevertheless has to be weighted against the cost of possible inconsistencies in the current and future interactions of the parties due to the discrepancy between the contract versions.

5 Related Work

The term 'contract' and the approach of introducing contracts in software components design stems from the Eiffel language [6], [8]; the core ideas of that work have greatly influenced our approach.

There are a number of works discussing the introduction of adapters between interacting parties to ensure their interoperability: [9], [10], [3], [11], [12], and [4] among others. Of specific interest to us is the work in [13], since they also make a clear distinction between the service producer and service consumer interfaces and protocols and use mappings to bridge them. Then they proceed to describe how to semi-automatically identify and resolve incompatibilities (mismatches) on interface and protocol level. Our approach extends this idea of separating producer and consumer specifications, but discusses how to avoid mismatches altogether instead of resolving them.

Furthermore, the W3C Technical Architecture Group has published an editorial draft on the extensibility and versioning of XML-based languages [14]. Their findings build on a number of previously developed theories and techniques like [15], [16] and draw lessons from the HTML and HTTP standards. They show how compatibility can be defined in terms of set theory, using super-sets and subsets to ensure compatibility. Our approach follows a similar way in dealing with the issue of compatibility, but instead of allowing the direct producer/consumer interaction, it introduces the contract as an intermediary to further decouple them.

The notion of service mapping comes from the field of schema evolution, i.e., the ability to change deployed schemas - metadata structures formally describing complex artifacts such as databases [17],[18],[7], messages, application programs or workflows. Typical schemas thus include relational or object-oriented (OO) database schemas, conceptual ER or UML models, ontologies, XML schemas, software interfaces and workflow specifications. Effective support for schema evolution is challenging since schema changes may have to be propagated, correctly and efficiently, to instance data, views, applications and other dependent system components. Our approach provides the means to identify schema changes that do not result in propagation of changes.

6 Conclusions and Future Work

In the work presented in the previous sections we discuss an approach that allows for transparency in the evolution of a service as viewed from the perspective of both clients and providers, in the context of the loosely-coupled nature of the SOA paradigm. For that purpose we introduce the contract construct as the means to leverage the decoupling of the interacting parties. We present a contract constructing function that bridges the gap between service matching and service mapping. Following on, we build on contractual invariance and contractual evolution to show how to effectively deal with shallow changes to the service provider and client interaction - without the need for adaptation which may lead in turn to deep changes.

There are of course a number of issues that are briefly discussed by our approach that we plan to work on in the future. The matter of management of the contracts and its relationship to service governance mechanisms is the most important issue at hand, since it can provide further insights on the proposed solution. Furthermore, the binding function ϑ value selection policy has to been further investigated. Using a static selection policy can be very restricting; a balancing mechanism for example can be applied for a more dynamic approach, expressed for example by negotiation between the parties in deciding the terms of the contract. Such a negotiation process during the formulation of the contract could result in the offering of additional or more specialized functionalities by the producer and could add a feedback loop to the presented algorithm for contract formulation. A promising direction when it comes to the implementation of our approach is to see whether it is possible to use techniques like the mapping constraints and tools developed by the schema mapping community like ToMAS [7].

The preservation of interoperability enforced by our approach is only the foundation in discussing the evolution of the interaction of parties. Following on, we plan to investigate how we can build on this work to deal with deep changes and the propagation mechanisms that run through them. On the other hand, another of the limitations of this work, the focus on the structural aspect of the service specification has also to be investigated, and examined if it is possible to apply the same approach to business protocols and policy-related constraints.

References

1. Papazoglou, M.P.: The challenges of service evolution. In: Bellahsène, Z., Léonard, M. (eds.) CAiSE 2008. LNCS, vol. 5074, pp. 1–15. Springer, Heidelberg (2008)
2. Andrikopoulos, V., Benbernou, S., Papazoglou, M.P.: Managing the evolution of service specifications. In: Bellahsène, Z., Léonard, M. (eds.) CAiSE 2008. LNCS, vol. 5074, pp. 359–374. Springer, Heidelberg (2008)
3. Ponnekanti, S.R., Fox, A.: Interoperability among independently evolving web services, Toronto, Canada, pp. 331–351. Springer, New York (2004)
4. Senivongse, T.: Enabling flexible cross-version interoperability for distributed services, p. 201. IEEE Computer Society, Los Alamitos (1999)
5. Papazoglou, M.P.: Web Service: Principles and Technology. Prentice Hall/Addison-Wesley (E) (2007)
6. Meyer, B.: Applying "design by contract". Computer 25, 40–51 (1992)
7. Velegrakis, Y., Miller, R.J., Popa, L., Mylopoulos, J.: Tomas: A system for adapting mappings while schemas evolve. In: ICDE, p. 862 (2004)
8. Meyer, B.: Object-Oriented Software Construction, 2nd edn. Prentice Hall PTR, Upper Saddle River (1997)
9. Yellin, D.M., Strom, R.E.: Protocol specifications and component adaptors. ACM Trans. Program. Lang. Syst. 19, 292–333 (1997)
10. Evans, H., Dickman, P.: Drastic: A runtime architecture for evolving, distributed, persistent systems. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 243–275. Springer, Heidelberg (1997)
11. Benatallah, B., Casati, F., Grigori, D., Nezhad, H.R.M., Toumani, F.: Developing adapters for web services integration. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 415–429. Springer, Heidelberg (2005)
12. Kongdenfha, W., Saint-Paul, R., Benatallah, B., Casati, F.: An aspect-oriented framework for service adaptation, 15–26 (2006)
13. Nezhad, H.R.M., Benatallah, B., Martens, A., Curbera, F., Casati, F.: Semi-automated adaptation of service interactions, Banff, Alberta, Canada, pp. 993–1002. ACM, New York (2007)
14. Orchard, D.(ed.): Extending and versioning languages: Terminology. W3C Technical Architecture Group (2007)
15. Orchard, D.: A theory of compatible versions. Published: xml.com article (2006)
16. Hoylen, S.(ed.): Xml schema versioning use cases, Published: W3C XML Schema Working Group Draft (2006)
17. Miller, R.J.: Retrospective on clio: Schema mapping and data exchange in practice. In: Description Logics (2007)
18. Fuxman, A., Hernández, M.A., Ho, C.T.H., Miller, R.J., Papotti, P., Popa, L.: Nested mappings: Schema mapping reloaded. In: VLDB, pp. 67–78 (2006)