Managing the Evolution of Service Specifications*

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

[1] INFOLAB, Dept. of Information Systems and Management, Tilburg University, The Netherlands
[2] LIRIS, Université de Lyon 1, France
{v.andrikopoulos,mikep}@uvt.nl, sbenbern@liris.univ-lyon1.fr

Abstract. The ability to cope with multiple competing stakeholders, fluid requirements, emergent behavior, and susceptibility to external pressures that can cause changes across an entire organization, coupled with the ability to support service diversification, is a key to an enterprise's competitiveness. Web services equip enterprises with the potential to react to change by addressing two interrelated sets of requirements: the ability to accommodate service changes that demand rapid response and to support service variation according to customers' needs and requirements. In this paper we introduce the concept of service evolution management, which provides an understanding of change impact, service changes control, tracking and auditing of service versions, and status accounting. To achieve this, we develop a formal model and theory for service evolution that allows multiple active service versions to be created consistently and co-exist, while executing schema changes effectively.

Keywords: Web services, service versioning, service differentiation, service contracts.

1 Introduction

XML-(or Web)-based services are key technologies providing a foundation for a net-centric services environment, which reacts to change by addressing two interrelated sets of requirements: the ability to accommodate service changes that demand rapid response, and the ability to support *service variation* according to the needs and requirements of customers. These two inter-related sets of requirements place emphasis on the ability of services to co-exist in *multiple active versions* and to execute changes effectively and efficiently. They therefore epitomize the common need for constant change that challenges service applications development. Service changes may, for instance, originate from the introduction of new functionality, the modification of existing functionality to improve performance, or the inclusion of new regulatory constraints that require that the

* The research leading to these results has received funding from the European Community's Seventh Framework Programme under the Network of Excellence S-Cube - Grant Agreement n° 215483.

Z. Bellahsène and M. Léonard (Eds.): CAiSE 2008, LNCS 5074, pp. 359–374, 2008.

behavior services is altered. Such changes should not be disruptive by requiring radical modifications in the very fabric of services or the way that business is conducted.

Service evolution is a precursor to successful service adaptation. Service adaptation refers to the *a posteriori* ability of a service to modify itself in order to interact with other services by detecting potential functional or non-functional mismatches with its peer services by semi-automated means ([1], [2]). Current service adaptation approaches assume that services can evolve independently and do not constrain their mutual inter-dependencies. In contrast to this, service evolution attempts to *a priori* validate and constrain service changes and ensuing service versions, so that they are consistent and well-behaved.

Routine change increases the propensity for error. To control service development one needs to know why a change was made, what are its implications and whether the change is complete. In a Web services environment, changes only affect the Web service provider's system. Typically Web service consumers do not immediately perceive the upgraded process, particularly the detailed changes of Web services. Hence, Web service based applications may fail on the Web service client side due to changes carried out during the provider service upgrade. In order to manage changes as a whole, the Web service consumers have to be taken into consideration as well, otherwise changes that are introduced at the service producer side can create severe disruption. Eliminating spurious results and inconsistencies that may occur due to uncontrolled changes is therefore a necessary condition for the ability of services to evolve gracefully, ensure service stability, and handle variability in their behavior. Thus, any service evolution management system has to be able to handle consistently and unambiguously the *propagation*, *validation*, and *conformance* of any kind of modifications applicable to a service.

Service evolution management requires an understanding of all the points of change impact, controlling service changes, tracking and auditing all service versions, and providing status accounting. In summary, service evolution management exhibits the following characteristics:

- **identification** of all kinds of permissible changes to services and **classification** of these changes,
- **propagation analysis mechanisms** that record the status of services, analyze changes, and gather information about their effects on clients of a service version,
- **validation and conformance mechanisms** that maintain the consistency of a service by ensuring that the service is a well-behaved collection of service changes and versions, and ensure conformance with respect to service updates and version contracts,
- **version control mechanisms** that control the release of a service and the changes applied to it throughout its lifecycle, and
- **instance migration mechanisms** for associating instances of running services with new service versions.

In this paper we shall consider all above items except for the topic of instance migration. This issue is examined by the workflow community (see section 2).

The paper is organized as follows: section 2 discusses related work from a number of different fields. In section 3 we present a service specification model that acts as a reference point in the discussion about service evolution. Service evolution management characteristics are covered in section 4. Finally, we wrap up the paper with some conclusions and future work (section 5).

2 Related Work

As services grow more complex to compensate for increasing business needs, valuable lessons and techniques can be drawn from Software Configuration Management (SCM), the discipline of software engineering that deals with controlling the evolution of complex software systems [3]. More specifically, the usage of *versions* as a representation of incrementally changed software objects (in that case, services) can be especially useful. The graph models that support the various versioning schemes [4] provide an intuitive way to manage the history of different versions.

Current Web services technologies do not directly address the versioning issue, usually requiring developers to solve the problem through the application of patterns and best practices [5]. Nevertheless, elements of these techniques can be used for service evolution management, see, for example, [6], [7], [8], and [9]. The common denominator of all these approaches is that they discuss *how* to put a versioning mechanism in place using an existing set of technologies, without concerning themselves with *what* constitutes the version of a service. An application of versioning in XML Schema technology is investigated in [10], where a number of use cases are presented that describe the desirable behaviors for XML Schema versioning. This approach provides guidelines for the behavior of schema processors in face of different versions, but it deliberately avoids discussing implementation (which is the critical component of that approach) and does not guarantee change consistency.

Lessons can also be drawn from the work on heterogeneous databases in general, and in specific from *schema mappings* between disparate data sources. Changes to the schemas of the data sources have to be reflected to the mappings between them. In that case the mappings have to be adapted to compensate for the evolution of the original source material [11], [12]. The evolution of requirements in information systems, as examined in the requirements engineering domain, is also a source of useful ideas and techniques. In [13] for example, a similar to ours approach is presented that combines abstraction from specific models with a generic typology for gaps (and similarities) in order to express evolution requirements.

Finally, we can also draw on the work of evolution in the field of workflows for methodologies and ideas. The problem of workflow evolution has two facets: *static*, referring to the issue of modifying the workflow description, and *dynamic*, referring to the problem of managing running instances of a workflow whose description has been modified (instance migration). The work in this field, at least in its conception, draws heavily from the literature on o-o databases evolution for its static aspects e.g. [14], but focuses mainly on the dynamic aspect [15], [16], [17].

3 Service Specification Reference Model

To be able to identify and study the changes happening to a service during its lifetime we can either choose a specific set of technologies for service description, like for example WSDL and BPEL, or use a technology-agnostic model that could easily be translated to the current standards. To abstract away from the idiosyncracies and syntactic nuances of these standards we chose the second approach. In particular, we introduce a *Service Specification Reference Model* that exhibits the main characteristics of different service description models and technologies.

More specifically, we define the following three levels for services specification:

Abstract Service Definition Model (ASD): an abstract model containing generic concepts and their relationships that are common to all service schemas.

Service Schema Definition (SSD): the schema of a specific service, or in other words, the service specification. This consists of the elements of the service and their relationships that are *generated* by corresponding concepts and relationships found in the ASD.

Instance of Service Schema Definition (ISD): that is produced from the instantiation of the SSD during the execution of the service.

Example 1 (Running Example). Consider the case of an (aggregate) service implementing a composite Order-to-Cash (OtC) process. OtC takes care of the revenue collection after a successful sale and shipping of a product. It involves a number of sub-processes/steps like purchase order processing, advanced shipping and delivery notification, invoicing, etc. These sub-processes are themselves exposed as services, offering their functionality to a number of services apart from the OtC. In that case, the specifications of all of the above services, expressed for example in WSDL, BPEL, etc., are considered the SSDs of each service. Irrespective of which set of standards is used for their description, all these services share a common reference framework that allows them to interoperate. All of them for example, allow clients to invoke them - or they are able to invoke back clients. These fundamental assumptions about how the services work constitute the ASD. Furthermore, when these services are implemented, deployed and invoked by their clients, then a number of instances of them are created based on their specification (their ISDs).

The following sections discuss only the ASD and SSD, starting with the general relationships that connect the building blocks of each level.

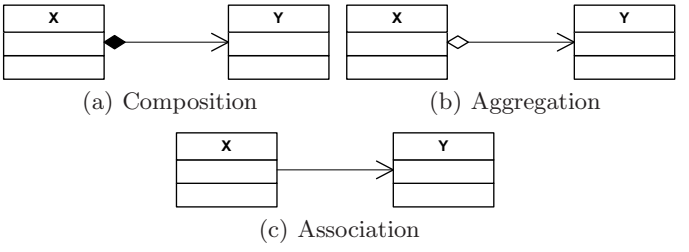
3.1 Universal Relationships

In order to show relationships between concepts in the ASD and therefore also between corresponding elements in the SSD we need to define the formal semantics of conventional relationships such as *composition*, *aggregation*, and *association* found in object-oriented languages and in the AI semantic nets. These will be described using the UML class diagram notation [18]:

Composition (fig. 1(a)) $\forall y, \exists! x : \overset{c}{\overrightarrow{xy}}$: y can belong in exactly one composition relationship with x. Additionally, deleting x deletes also y (*cascading delete*).

Aggregation (fig. 1(b)) $\forall y, \exists x : \overset{a}{\overrightarrow{xy}}$: y may participate in more than one aggregation relationships with x. Deletion of x deletes also y, but only if there are no other relationships of this type in which y participates.

Association (fig. 1(c)) $\exists y, \exists x : \overset{s}{\overrightarrow{xy}}$: No further restrictions on the participation and the existence of y. We use the notation $\overset{r}{\overrightarrow{xy}}, r \in \{c, a, s\}$ to show that elements x and y have a relationship of type r.



(a) Composition (b) Aggregation

(c) Association

Fig. 1. Types of Relationships

Furthermore, relationships in UML have multiplicities, denoting the possible *cardinalities* of the instances of each class, that is, how many instances of it may exist at the same time, in each relationship. Putting a *1..** on the y side of the arrow for the relationship $\overset{s}{\overrightarrow{xy}}$ for example means that instance x must always have one or more associations with instances of y. The notation we introduced above to show the semantics of the relationships has therefore to be expanded to accommodate cardinalities with: $|cardinality_x||cardinality_y|^{-1}, cardinality \in \{0| \geq 0| \leq 1| \geq 1\}$ (≥ 0 for $*$ in UML, ≤ 1 for *0..1*, and ≥ 1 for *1..** respectively). For example, $\overset{a}{\overrightarrow{xy}}\ |1||\geq 1|^{-1}$ means there may be more than one instances of y that participate in this aggregation with x. If not specified explicitly, cardinality values are considered to be equal to 1.

3.2 Abstract Service Definition Model

The Abstract Service Definition Model (ASD) is a collection of *concepts* common across service schemas that have a set of *parameters* and *relationships*. Parameters can either be *property-domains* or *attributes*. An attribute, e.g. a string denoting the currency that will be used in the scope of a specific message, will be assigned a value during the *instantiation* of the service schema, i.e. the creation of a running instance of the service from its schema. Property-domain pd_i has a set of values called properties (denoted by p_i) that, as we will see in section 3.3, may restrict some of the relationships of the element generated by the concept. A specific value p_i will be selected from the domain when the SSD is generated by the ASD.

The ASD spans three layers: *Structural*, *Behavioral*, and *Regulatory*, which lean on each other in ascending order to fulfill their functionality, and two sections: *Public* and *Private* (denoted by a *visibility* attribute of the concept definition). Figure 2 illustrates the ASD using UML class diagram notation.

ASD Notation. The concepts depicted in the figure and their relationships are partly based on the models discussed in [19] and [20], which describe metamodels for services. Attributes are not contained in the figure for reasons of brevity.

Based on the above, the concepts in Figure 2 can be described using the following notation: $Concept(property-domain^*, attribute^*, relation^*, visibility)$.

Example 2. The `Message` concept as shown in this figure, has two property-domains: `messageRole`, and `type` (that is inherited from concept `Role`[1]), and an one-to-many aggregation relationship with the `Information Type` concept.

Consequently, it can be written in this notation as: $Message(messageRole, type, attributes^*, \overrightarrow{Message\ InformationType}^{a} |1|| \geq 1|^{-1}, public)$.

Definition 1. ***ASD Layers*** *The ASD concepts can be perceived horizontally as in the three distinct layers, viz. structural, behavioral and regulatory (see Figure 2). Each layer $L_i, i = 1, 2, 3$ is defined as $L_i = \{\exists! \mathcal{P}_x \in ASD / \forall x \in \mathcal{P}_x \rightarrow x \in L_i\}$, where $\mathcal{P}_x$ is a* partitioning *of all concepts into exactly one of the three layers. Now we can define the notion of* horizontal *and* vertical *relationships based on this partitioning:*

Horizontal: $[[\overrightarrow{xy}]]^{r^h}_k : x, y \in L_k$*, i.e., all concepts belong to the same layer.*

Vertical: $[[\overrightarrow{xy}]]^{r^v}_k : x \in L_k, y \in L_m, m \neq k$*, i.e. concepts are in different layers.*

For example, the `Policy Profile` concept has a horizontal relationship with `Policy Alternative` and `Service Policy` since they belong to the same layer (the regulatory), and a vertical relationship with the `Operation Sequence` and `Operation` concepts in the behavioral and structural layers respectively (see Figure 2).

Definition 2. ***The ASD.*** *The* $ASD = \{(c^j_i, L_i), i \leq 3, j \geq 1$ *and* $\forall c^n_l \ |\overrightarrow{c^j_i c^n_l}^{r}, (i = l \wedge r = r^h) \vee (i \neq l \wedge r = r^v)\}$ *where c^j_i is a concept c^j in the layer L_i, i.e. ASD is the set of all concepts and the layers they belong to.*

For this paper we have mainly concentrated on the structural and behavioral layers of the ASD.

ASD concepts. In the following we briefly present the layers and the concepts of the ASD:

[1] We assume that the inheritance relationship maintains its UML semantics, i.e., all attributes and parameters (but not relationships) are copied to the inherited concept.

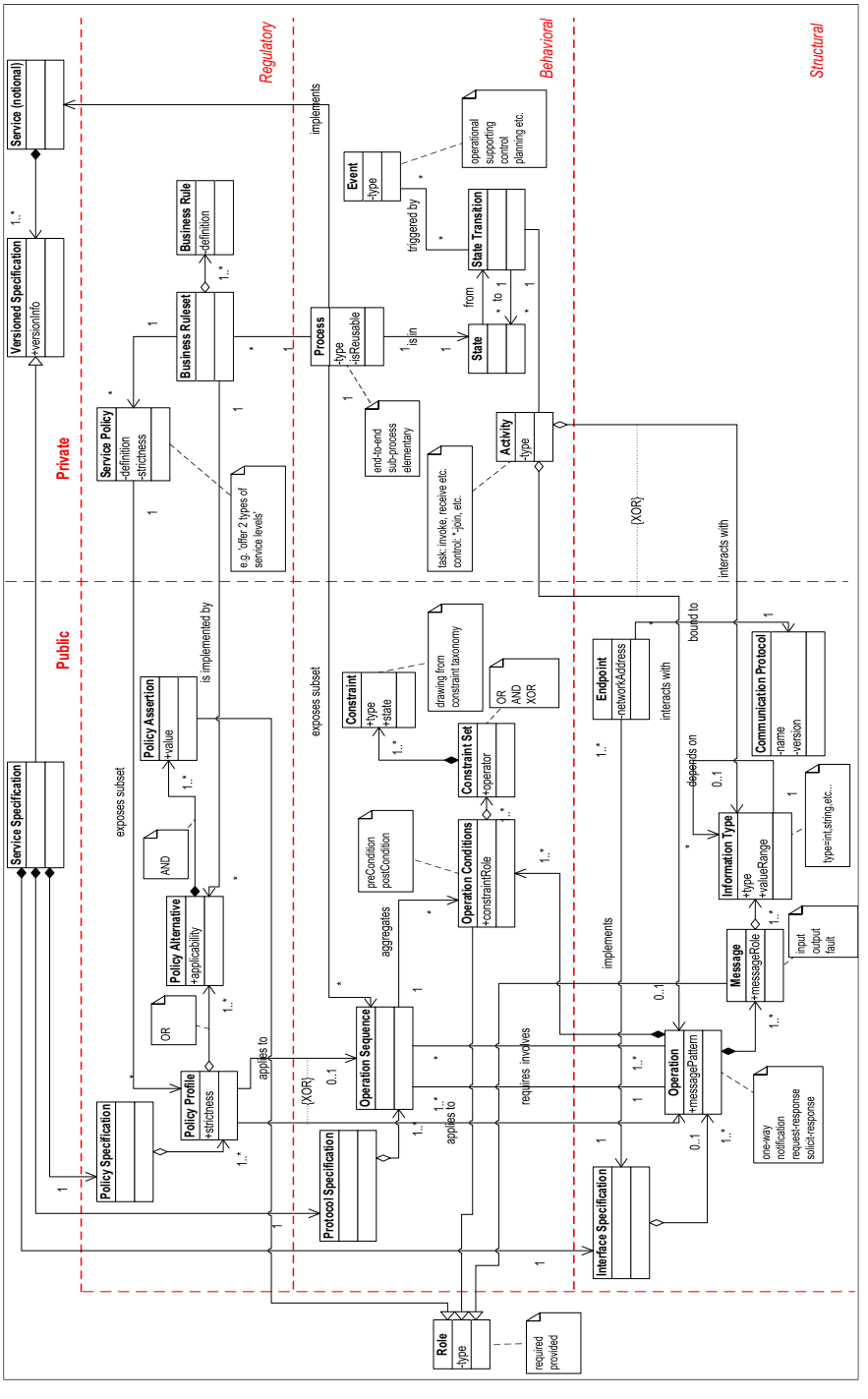


Fig. 2. Layers of the ASD

Structural layer is responsible for the structural description of the concepts that constitute a service. In that sense it contains: an `Interface Specification`, that acts as the access point for the structural signature of the service, aggregating a number of `Operation`s, i.e. specific functions that the service is performing, defined in an abstract format, and `Message`s that are exchanged as part of the operations. A `Message` is defining the informational content consumed and/or produced by the service in the form of hierarchically organized `Information Type`s. Furthermore, `Endpoint`s, define specific URIs that the service can be reached from. Each endpoint is bound to a number of `Communication Protocol`s that can be used to access it.

Behavioral layer manages the control and execution aspects of the service business logic. It contains for this purpose: a `Protocol Specification` that aggregates the necessary constructs for specifying business protocol information, i.e. `Operation Sequence`s (specific sequences that have to be performed in order to achieve a part of the business logic). They aggregate sets of `Operation Condition`s that either have to be fulfilled in order for the particular sequence to be executed, or are produced by it during its execution, in terms of pre- and post-executional `Constraint`s (e.g., temporal, spatial, financial, etc.). `Constraint`s are organized around `Constraint Set`s that may contain overlapping constraints. `Constraint Set`s and `Constraint`s are for this layer what `Message`s and `Information Type`s are for the Structural: a method to express the signature of services - but this time on protocol level. In order to model the workflow aspect of the service description, the concept of `State`s of the `Process` for which the service is the implementation is used. `State Transition`s govern the switch from one state to the other. The switch can either occur as the outcome of the successful execution of an `Activity` and/or triggered by an `Event`.

Regulatory layer contains the necessary elements for describing and managing both the business decision rules and policies that govern the business logic of the process, and key performance indicators, e.g., QoS factors, and requirements set by the service clients and providers. It contains: a `Policy Specification` that collects the various policies defined as part of the service operation. `Policy Profile`s define the business rules and/or the functional and non-functional requirements of complete `Operation Sequence`s or atomic `Operation`s through `Policy Alternative`s that are expressed as a number of `Policy Assertion`s. They implement a `Business Ruleset`, which is defined as a set of `Business Rule`s specified by the owner of the service. These rules constitute an abstract `Service Policy` that all versions of the service have to respect.

Auxiliary concepts Figure 2 also contains a number of auxiliary concepts that do not belong to a specific layer but are used throughout the ASD. The `Role` concept inherits a very important as we will see in later sections property-domain to a number of concepts. `Service (notional)` acts as the main reference point for all *versions* of the service, expressed as `Versioned Specification`s - `Service Specification`s that aggregate the specifications from each layer, enriched with versioning information.

Public and Private sections. A vertical distinction of concepts is necessary in order to distinguish between service elements exposed to the clients of the service (public specifications) and the private specifications that are solely used for internal service purposes. The former are the access points for service clients that want to interact with the specified service. The latter are used for example to define the workflow of the process implemented by the service, and the service policies that are common among all versions of the service. Each service uses a combination of public and private concepts to specify itself; it exposes to its context only the public ones, but at the same time it incorporates the public concepts of the services it uses in turn. This implies that a uniform representation model is used across the network of interrelated services.

3.3 Service Schema Definition

The Service Schema Definition (SSD) consists of a number of elements and their relationships. In the following we discuss the structure and the properties of the SSD.

Definition of service schema. The SSD (for a particular service) is generated from the ASD by evaluating the concepts in it, i.e., by assigning a uniquely identified name to them, deciding on a specific property p_i for each properties-domain pd_i, and instantiating their relationships by replacing concepts with elements and assigning a value to their cardinality. The SSD is defined as a set of *elements* $\mathcal{E} = \{e_i\}, i = 1, \ldots, n$ and each element e is described by a tuple as follows in BNF-format:

$$e := < name, attribute^*, (property^?, relation^?)^*, visibility >$$
$$name := string \quad property := string \quad attribute := string$$
$$relation := RT_e|c||c|^{-1}$$
$$RT_e := \overset{c}{\overrightarrow{ee_i}} \mid \overset{a}{\overrightarrow{ee_i}} \mid \overset{s}{\overrightarrow{ee_i}} \;\; where\ i \geq 1$$
$$c := integer \quad visibility := public|private$$

'*' means 0-n occurrences and '?' means 0-1 occurrences

Example 3. Consider on ASD level the `Operation` concept defined as[2]:

$Operation(messagePattern, Att^*, \overset{c}{\overrightarrow{Operation\ Message}}\ |1||\geq 1|^{-1}, public)$.

On SSD level, the evaluation of the concept for the creation of the SSD for the purchase order processing process from example 1 would e.g. generate:

$$e_1 = < val(Operation), val(messagePattern), Att^*,$$
$$val(\overset{c}{\overrightarrow{Operation\ Message}}\ |1||\geq 1|^{-1}), public >$$
$$= < processOrder, request - response, Att^*, \overset{c}{\overrightarrow{e_1e_2}}\ |1||1|^{-1}, \overset{c}{\overrightarrow{e_1e_3}}\ |1||1|^{-1}, public >$$
$$e_2 = < purchaseOrder, Att^*, input, \ldots, public >$$
$$e_3 = < orderAcknowledge, Att^*, output, \ldots, public >$$

[2] *Note:* we are omitting some of the relationships of the concept for brevity.

where $val(messagePattern) = request - response$ means that $request - response$ is a property in the property domain `messagePattern`.

Note: since e_1 was generated by `Operation`, it is said to be an `Operation` element. In the same manner e_2 and e_3 are `Message` elements. For the remainder of this paper we shall use this convention to identify elements by the concepts they are generated from. In the same spirit, the distinction of concepts into layers (definition 1) also applies to elements. Therefore e_1, e_2, e_3 are said to belong to the *Structural* layer.

Constraining the service schema elements. The properties defined previously may restrict *(a)* the cardinality of the relationship(s) of the element, and *(b)* some of the properties of the related element(s).

Some properties of an element therefore define a number of constraints. Let's assume $\mathcal{P}_x$ the set of the properties assigned to an element x, then $\exists p \in \mathcal{P}_x$, the relation $\overset{r}{\overrightarrow{xy}}$ must satisfy the following: $\begin{cases} |\ |^{-1} = k \\ \text{and zero or more of:} \\ pd_y = p_{y_x} \end{cases}$ where the values of the property domain pd_y are restricted by the properties p_{y_x} (a subset of the original properties domain p_y defined by the property x).

Example 4. Consider the relation between `Operation` and `Message` elements in the structural layer. `Operation` has a property domain $pd_1 =$ `messagePattern` that takes the following values/properties $\{one - way,$ $notification, request-response, solicit-response\}$, and `Message` has two property domains $pd_2 = message-role$ and $pd_3 = type$; the latter takes the property $\{required, provided\}$ and the former takes the properties $\{input, output, fault\}$. The property `one-way` defines the constraints:

$|\ |^{-1} = 1$, and $messageRole = input, type = required$, i.e. there can only be one `Message` element related to the `Operation` element, and it has to have the properties *input* and *required*. In a similar way, property `request-response` defines the constraints:

$$|\ |^{-1} = 2, \text{ and } \begin{cases} Msg_1.messageRole = input, Msg_1.type = required \\ Msg_2.messageRole = output, Msg_2.type = provided \end{cases}$$

or

$$|\ |^{-1} = 3, \text{ and } \begin{cases} \text{(as above)}, \\ Msg_3.messageRole = fault, Msg_3.type = provided \end{cases}$$

where Msg_1, Msg_2, and Msg_3 are occurrences of the `Message` element.

4 Service Schema Evolution

The evolution of the service is taking place through a series of discrete modifications to elements in its schema that constitute *evolutionary acts* in the sense that they are the carriers of change. These acts are expressed as sets of *operations* that may have consequences both inside and across the service.

4.1 Operations on the Service Schema

The following list is a minimal classification of operations and their semantics that we have identified, in the form of primitive changes on the elements of the SSD and/or their relationships.

1. *Insertion of Relationship between Elements* $[[ADD(e_i, e_j, r)]] : \mathcal{R}_{e_i} \rightarrow \mathcal{R}_{e_i} \cup \{\overset{r}{\overrightarrow{e_i e_j}}\}$, where $\mathcal{R}_{e_i}$ is the set of relationships of element e_i.
2. *Removal of Relationship between Elements* $[[DEL(e_i, e_j, r)]] : \mathcal{R}_{e_i} \rightarrow \mathcal{R}_{e_i} - \{\overset{r}{\overrightarrow{e_i e_j}}\}$. Bilateral relationships are deleted by performing this operation once for each direction.
3. *Insertion of Element* [[ADD(e)]]: $\mathcal{E} \rightarrow \mathcal{E} \cup \{e\}$ and/or not, $ADD(e_j, e, r)$ Insertion of an element e may be accompanied by the insertion of a relationship between another (preexisting) element e_j and e.
4. *Removal of Element* [[DEL(e)]]: $\mathcal{E} \rightarrow \mathcal{E} - \{e\}$ and $if\ \exists\ \overset{r_j}{\overrightarrow{e_j e}}\ \forall r_j \in \mathcal{R}, \forall j$ *then* $DEL(e_j, e, r_j)$,
   where $\mathcal{R} = \{\overset{r}{\overrightarrow{e_i e_j}}\ \forall i, \forall j, \forall r\}$the set of all relationships for all elements in $\mathcal{E}$. Removal of element e must always be preceded by the removal of all the relationships that e is participating in.
5. *Replacement of Property* [[REP(e,p_i, p_j)]]:$\mathcal{P}_e \rightarrow \mathcal{P}_e \cup \{p_j\} - \{p_i\}$ A property can either be replaced ($p_i, p_j \in pd_k$), added (for $p_i = \oslash$), or deleted by replacing it with an empty property ($p_j = \oslash$). In case that the properties constrain the cardinality of (some) relationships, then the appropriate operations on the relationships of e must be performed too.

This set of operations is *complete*; it can be easily shown that every SSD can be constructed from another SSD by a finite sequential application of additions, deletions, and replacements to it. In that sense, richer typologies of operations like the ones in [13], [17], and elsewhere, would be more convenient but not necessary for our approach.

4.2 Service Schema Versioning

The definition of SSD as a set of elements uniquely identified by their name which is subjected to a number of modifications allows us to draw on versioning techniques from Software Configuration Management. Based on the terminology used in [4], we define the spaces of a service:

Definition 3. ***Service spaces***

1. *The elements belonging to the SSD are the* product space *of the service, denoted by* ps*, which contains the specifications of the various versions of the service.*
2. *The temporal relationships between all the versions of the elements that constitute a service is called* version space *and is denoted by* vs*.*

3. *A version v^e represents the state of the evolving element* e *and it is characterized by the pair $v^e = (ps^e, vs^e)$, where ps^e denotes its state in the product space (i.e. the specification of the element), and vs^e its position in the version space (denoted by a version identifier). A version v^s of a service* s *is therefore defined as the set of the versions of its elements $v^s = \{v^e, \forall e \in \mathcal{E}\}$ at a given time (-point in the version space).*

Service versioning comprises service specifications as observed at discrete points in time. These are identifiable by a version identification number; each version is agnostic of the others and managed individually. Each of the service versions is created by applying a number of changes to a previous service version, which can be thought of as the *baseline* for that version. Information regarding the baseline of each version, and how a service version differs from its baseline constitutes the *version history* of a given service. There might be, for instance, three versions of the SSD (signified by the version ids '2.1', '2.1.1', and '2.1.2'). Each version is a full-fledged service schema specification and corresponds to different (possible) active versions of the service. By examining the version space of the service, the designer is able to infer that versions '2.1.1' and '2.1.2' use '2.1' as a baseline and additionally what changes where applied to '2.1' in order to produce these two versions. In that respect, the *extensional versioning* scheme is used to record version history and is defined as follows:

Definition 4. ***Extensional versioning***

1. *Let's assume a sequence of elementary change operations $op_1 \dots op_m$ which, when applied to one version of an element v^e, v_i^e, yields another version v_j^e, denoted as $v_i^e \triangleright v_j^e$.*
2. *The extensional versioning of an element is the set V^e of all versions of* e *and is defined by $V^e = \{v_1^e, v_1^e \triangleright v_2^e, v_2^e \triangleright v_3^e \dots, v_m^e \triangleright v_n^e\}, m < n$. We therefore keep track of all versions with the corresponding operation changes. By extension then, the set of all versions of a service, is defined as $V^s = \{v_1^s, \dots, v_n^s\}$.*
3. *An* evolutionary act *can therefore be defined as the set of operations that transform version v_i^s of the service into version v_j^s and is denoted by $v_i^s \triangleright v_j^s, i < j$.*

4.3 Consistency of Service Schema Evolution

The operations and the versioning approach presented in the previous section are generic enough to cover all possible modifications to service elements; but they do not make any effort to ensure that these modifications are meaningful - that is, they are not destructive for the SSD. For example, the *DEL(e)* operation removes an element completely from the current version of a service; but is this a valid operation for all elements of the SSD? In order to preserve the consistency of an SSD it is necessary to define a set of *Invariants*, such as those defined in [21]. These invariants (must) hold at every state of the SSD and ensure that the SSD is never left in an *inconsistent* state (i.e. a state that violates any invariant):

$\mathcal{INV}_1$**: Validity of the SSD.** $\mathcal{ASD} \models \mathcal{SSD}$: the SSD must always be *valid* with respect to the ASD, i.e., every element and relationship in the SSD must be able to be generated from the respective ASD concepts and relationships. This also includes the preservation of the semantics of the relationships, as defined in section 3.1.

$\mathcal{INV}_2$**: Reachability of Elements.** $\forall e \in \mathcal{E}$, *then* $\exists e_j \in \mathcal{E}$, $\exists r \in \mathcal{R}$, $\overset{r}{\overrightarrow{ee_j}}$: All elements must participate in at least one (directed) relationship with another element. If there are elements without any relationships in the schema then they are automatically deleted.

$\mathcal{INV}_3$**: Cardinality Constraint Preservation.** $\exists p \in \mathcal{P}_e, \exists \overset{r}{\overrightarrow{ee_j}}$ *with* $|j|^{-1} = k$ *then* $\overset{r}{\overrightarrow{ee_j}} \triangleright \overset{r}{\overrightarrow{ee_{j'}}}, |j'|^{-1} = k$: If there is a property of the element that constraints the cardinality of some relationship of the element, then this constraint must be respected by all versions of the element.

$\mathcal{INV}_4$**: Existence Constraint on Composition.** $\forall e \in \mathcal{E}$ *if* $\overset{c}{\overrightarrow{ee_j}}$ *and* $DEL(e)$ *then* $DEL(e_j), \forall j$: If an element with composition relationships is deleted, then all its related elements through composition must be deleted too. (*Note:* The case of aggregation is covered by $\mathcal{INV}_1$.)

Now we can define the notions of *consistency* and *consistent evolutionary acts*:

Definition 5. *A version of the SSD is called Consistent iff it respects the set of obligatory invariants* $\mathcal{INV}^o = \{\mathcal{INV}_i, 1 \leq i \leq 4\}$. *Consistent evolutionary acts are therefore the series of operations that preserve the consistency of the SSD.*

For example, reducing the payload of a `Message` element by deleting one of the `Information Type` elements that it is related to is considered consistent. Deleting *all* of the `Information Type`s though is inconsistent, since it violates $\mathcal{INV}_1$; as shown in Figure 2, this relationship must have cardinality *at least* 1. The former then is a consistent evolutionary act, the latter isn't.

4.4 Conformance of Service Schema Versions

In summary, consistency ensures that the evolutionary acts are valid transformations of one version of the service SSD into another version. Taking into account the fact that services work in a network environment, using each other to achieve their stated objectives, creates the added necessity for the *preservation* of the service execution result. This ensures the seamless substitution of an SSD by a new version of it, without requiring any modifications by the clients of the service (its context); in other words, the *conformance* of the two versions in terms of expected results of service execution and not (only) in terms of specification:

Definition 6. *Given two consistent versions* v_i^s *and* v_j^s *of a service, they are called Conformant iff* v_i^s, v_j^s *can be interchangeable without requiring changes in their context.*

Example 5. Consider the case of the owner of the invoicing service wanting to expand its operations to international marketplaces. For that purpose, a new

version of the service is created. Among other changes, information is added to the invoice data schema about the currency that the payments are to be made in, the tax regulations that apply to the specific invoice, etc. As long as the existing clients of the service can still use the same service specification by simply ignoring this additional information, the two service versions are considered to be conformant with respect to this change.

We have identified the following invariants that could ensure conformance:

$\mathcal{INV}_5$**: Co-Variance of Required Elements.** All elements that have the property `required` can only be *restricted*. This implies a restriction in the data type and the number of arguments (represented as relationships between elements). It can be stated as follows:

- in the number of relationships: $\exists p \in \mathcal{P}_e = required\ and\ if \exists\ \overset{r}{\overrightarrow{ee_j}}\ then\ \overset{r}{\overrightarrow{ee_j}} \triangleright \overset{r}{\overrightarrow{ee_j\prime}}, |j\prime|^{-1} \leq |j|^{-1}$, i.e. the element must have the same number or less relationships after any change to it,
- in the value domain of its properties: if the property is defined as a range of values, then this range can only be *restricted*.

This also holds for all elements in $\mathcal{R}_e$, the set of all relationships of e.

$\mathcal{INV}_6$**: Contra-Variance of Provided Elements.** All elements that have the property `provided` can only be *extended*. This implies an extension in the data type and the number of arguments. It can be stated as follows:

- in the number of relationships: $\exists p \in \mathcal{P}_e = provided\ and\ if \exists\ \overset{r}{\overrightarrow{ee_j}}\ then\ \overset{r}{\overrightarrow{ee_j}} \triangleright \overset{r}{\overrightarrow{ee_j\prime}}, |j\prime|^{-1} \geq |j|^{-1}$, i.e. the element must have the same number or more relationships after any change to it,
- in the value domain of its properties: if the property is defined as a range of values, then this range can only be *expanded*.

This also holds for all elements in $\mathcal{R}_e$.

$\mathcal{INV}_7$**: Finality of Cardinality-Constraining Properties.** The properties that constraint the cardinality of (some) relationship of a given service element are *final*, i.e. no such property of any element is allowed to be modified. Properties with no constraints on relationships can be subjects of change as long they respect the previous invariants.

For example, increasing the number of `Constraints` in a `Constraint Set` (see Figure 2) is only allowed if it is related to an `Operation Conditions` element that has the property `provided`, but not if it has the property `required`. The same applies also to the property `valueRange` in `Information Type`: the property can only be replaced by a 'smaller' range in the former case, and by a 'wider' one in the latter. Modifying the `MessagePattern` property in any way is forbidden since it constraints the cardinality of the `Operation` element with element(s) `Message`. Therefore:

Definition 7. *Conformance preservation is the property of an evolutionary act to respect the set* $\mathcal{INV}^c = \{\mathcal{INV}_i, 5 \leq i \leq 7\}$*. Conformance-preserving evolutionary acts therefore create conformant versions of the service.*

Example 6. Assume that the OtC service described in example 1 is used by a Purchase-to-Pay (PtP) process of another enterprise. PtP takes care of the procurement of goods process and at a certain step uses the invoice produced by the OtC to arrange for payments. In that case, the same invoice document (more accurately, the `Information Type` element that corresponds to the invoice) is a provided element for OtC and according to $\mathcal{INV}_6$ it can be extended in the manner described above. However, since PtP uses the same element as an input (and therefore it is required for it), then $\mathcal{INV}_5$ forbids this modification. In that case, the PtP service can not use the new version of OtC and has to rely on the previous one to do business (ensuring in that way that there are no misunderstandings in the currency that the transactions take place).

What is illustrated by the previous example is the fact that $\mathcal{INV}^c$ is a set of *necessary* but not *sufficient* conditions for conformance. That is a by-product of the loosely-coupled nature of the service-oriented architecture: it is not desirable to be able to reason explicitly about the effect of a service change at provider side to its client services. This is due to the fact that client services should be oblivious to the changes that happen to a provider service. This enforces a modus operandi based on the separation of concerns: each party will decide from their own perspective whether a new version and the evolutionary act that created it is conformance-preserving with respect to their own services. In that sense, two services using each other have an implicit *contract* between them, enforced partially by each side using their interpretation of $\mathcal{INV}^c$. Every new version that is issued by a service proposes the alteration of this contract between them; it is up to the other party to decide whether the proposed changes are acceptable or not.

5 Conclusions and Future Work

In this paper we have introduced service evolution management facilities that identify and classify all kinds of permissible changes to services, analyze the propagation effects of changes, introduce version control mechanisms, validate the completeness of a change, and maintain consistency by ensuring that a service is a well-behaved collection of service changes and versions. The service evolution management facilities rely on a service specification reference model that abstracts away from the idiosyncracies and syntactic nuances of current standards and provides a theoretical approach to service evolution. The service specification reference model contains an abstract service definition model (ASD) that comprises generic concepts and inter-relationships that are common to all service schemas in three layers. Thus far, we have concentrated on representing and analyzing the behavior of multiple active service versions that are mutually conformant with respect to a contract from both the perspective of the service provider and the service client. In the future, we expect to concentrate on developing formalisms and proofs for the service regulatory layer and connect them with current work, so as to be able to prove the completeness and soundness of the overall approach. Another extension of this work is to focus on relaxed co- and contra-variance mechanisms for more flexible service evolution purposes, e.g., exception handling.

References

1. Ponnekanti, S., Fox, A.: Interoperability among independently evolving web services. In: Middleware, pp. 331–351 (2004)
2. Benatallah, B., Casati, F., Grigori, D., Nezhad, H.R.M., Toumani, F.: Developing adapters for web services integration. In: CAiSE, pp. 415–429 (2005)
3. Tichy, W.F.: Tools for software configuration management. In: SCM, pp. 1–20 (1988)
4. Conradi, R., Westfechtel, B.: Version models for software configuration management. ACM Comput. Surv. 30(2), 232–282 (1998)
5. Brown, K., Ellis, M.: Best practices for Web services versioning. IBM developerWorks White Paper (2005)
6. Russell, M.: Manage message contract changes with versioning. IBM developerWorks White Paper (2005)
7. Butek, R.: Make minor backward-compatible changes to your Web services. IBM developerWorks White Paper (2004)
8. Poulin, M.: Service Versioning For SOA. SOAWorld Magazine 6(7) (2006)
9. Kaminski, P., Litoiu, M., Müller, H.A.: A design technique for evolving web services. In: CASCON, pp. 303–317 (2006)
10. Hoylen, S.(ed.): XML Schema Versioning Use Cases. W3C XML Schema Working Group Draft (2006)
11. Velegrakis, Y., Miller, R.J., Popa, L.: Mapping adaptation under evolving schemas. In: VLDB 2003: Proceedings of the 29th international conference on Very large data bases, VLDB Endowment, pp. 584–595 (2003)
12. Yu, C., Popa, L.: Semantic adaptation of schema mappings when schemas evolve. In: VLDB 2005: Proceedings of the 31st international conference on Very large data bases, VLDB Endowment, pp. 1006–1017 (2005)
13. Salinesi, C., Etien, A., Zoukar, I.: A Systematic Approach to Express IS Evolution Requirements Using Gap Modelling and Similarity Modelling Techniques. In: Persson, A., Stirna, J. (eds.) CAiSE 2004. LNCS, vol. 3084, pp. 338–352. Springer, Heidelberg (2004)
14. Casati, F., Ceri, S., Pernici, B., Pozzi, G.: Workflow evolution. In: Thalheim, B. (ed.) ER 1996. LNCS, vol. 1157, pp. 438–455. Springer, London (1996)
15. Reichert, M., Dadam, P.: ADEPTflex - supporting dynamic changes of workflows without losing control. J. Intell. Inf. Syst. 10(2), 93–129 (1998)
16. Joeris, G., Herzog, O.: Managing evolving workflow specifications with schema versioning and migration rules (1999)
17. Weber, B., Rinderle, S., Reichert, M.: Change Patterns and Change Support Features in Process-Aware Information Systems. In: Krogstie, J., Opdahl, A., Sindre, G. (eds.) CAiSE 2007 and WES 2007. LNCS, vol. 4495, pp. 574–588. Springer, Heidelberg (2007)
18. Rumbaugh, J., Jacobson, I., Booch, G.: Unified Modeling Language Reference Manual, 2nd edn. Addison-Wesley Object Technology Series. Addison-Wesley Professional, Reading (2004)
19. Everware-CBDI Inc.: CBDI-SAE^{TM} Meta Model for SOA Version 2.0. (2007), `http://www.cbdiforum.com/public/meta_model_v2.php`
20. Dubray, J.J.: WSPER An abstract SOA framework (2007), `http://www.wsper.org/primer.html`
21. Banerjee, J., Kim, W., Kim, H.J., Korth, H.F.: Semantics and implementation of schema evolution in object-oriented databases. In: SIGMOD 1987: Proceedings of the 1987 ACM SIGMOD international conference on Management of data, pp. 311–322. ACM Press, New York (1987)