Automatically Determining Compatibility of Evolving Services

Karin Becker, Andre Lopes, Dejan Milojicic, Jim Pruyne, Sharad Singhal
HP Laboratories

HPL-2008-49
May 21, 2008*

SOA, service, backward compatibility, version

A major advantage of Service-Oriented Architectures (SOA) is composition and coordination of loosely coupled services. Because the development lifecycles of services and clients are de-coupled, multiple service versions have to be maintained to continue supporting older clients. Typically versions are managed within the SOA by updating service descriptions using conventions on version numbers and namespaces. In all cases, the compatibility among services description must be evaluated, which can be hard, error-prone and costly if performed manually, particularly for complex descriptions. In this paper, we describe a method to automatically determine when two service descriptions are backward compatible. We then describe a case study to illustrate how we leveraged version compatibility information in a SOA environment and present initial performance overheads of doing so. By automatically exploring compatibility information, a) service developers can assess the impact of proposed changes; b) proper versioning requirements can be put in client implementations guaranteeing that incompatibilities will not occur during run-time; and c) messages exchanged in the SOA can be validated to ensure that only expected messages or compatible ones are exchanged.

Submitted to the IEEE International Conference one Web Services (ICWS), 2008 China

Automatically Determining Compatibility of Evolving Services

Karin Becker, Andre Lopes
T&T Engenheiros Associados Ltda.
{karin.becker, andre.lopes}@hp.com

Dejan Milojicic, Jim Pruyne, Sharad Singhal
HP Labs
{dejan.milojicic,jim.pruyne,sharad.singhal}@hp.com

Abstract

A major advantage of Service-Oriented Architectures (SOA) is composition and coordination of loosely coupled services. Because the development lifecycles of services and clients are de-coupled, multiple service versions have to be maintained to continue supporting older clients. Typically versions are managed within the SOA by updating service descriptions using conventions on version numbers and namespaces. In all cases, the compatibility among services description must be evaluated, which can be hard, error-prone and costly if performed manually, particularly for complex descriptions. In this paper, we describe a method to automatically determine when two service descriptions are backward compatible. We then describe a case study to illustrate how we leveraged version compatibility information in a SOA environment and present initial performance overheads of doing so. By automatically exploring compatibility information, a) service developers can assess the impact of proposed changes; b) proper versioning requirements can be put in client implementations guaranteeing that incompatibilities will not occur during run-time; and c) messages exchanged in the SOA can be validated to ensure that only expected messages or compatible ones are exchanged.

1. Introduction

One of the major advantages claimed for web services, and in general for service-oriented architecture (SOA), is the ease of making changes. SOA is an architectural paradigm that supports the usage, composition and coordination of autonomous, sharable services in a loosely coupled manner. SOA enables independent development by disparate teams, each one with its own delivery and maintenance schedule [8]. The decoupled life-cycles of services and clients have major consequences from a change management perspective. As a service is upgraded, it must continue to support existing clients. Likewise, it must also support newer clients that desire to use new or improved features. This requires the ability to represent and manage multiple versions of the same service within the SOA, and transparently enable redirection of old clients to the new versions of the service when possible. Ideally, compatible changes should not cause failures or unexpected behavior. Hence, research is required in how changes are introduced in services within the SOA [1], [5], [6], [8], [9]. Finally, because every service can be used in multiple solutions, any change in the behavior of a service can cascade across several clients (and clients of those clients) in a transitive manner, causing a broad impact within the SOA. Thus the SOA must provide capabilities that allow early detection of incompatible changes.

Currently, there is no comprehensive solution for managing compatibility between service versions in SOA. Existing work can be divided into: a) best practices and design patterns for service versioning (e.g. [1], [2], [5], [10], [13]), b) version-aware registry solutions (e.g. UDDI v3.0.2, Systinet, [7]), which make assumptions about service compatibility, and c) architectural components for dealing with service compatibility within the SOA (e.g. [6], [8], [9]). In all cases, it is assumed that compatibility between versions is assessed manually, a particularly error-prone task for complex service descriptions.

In this paper, we describe an approach for automatically assessing service compatibility between related versions of a service. Unlike related approaches, we do not infer compatibility from version number conventions, nor do we restrict changes between service versions to avoid incompatibility. The compatibility assessment method proposed is based on a version framework that allows service descriptions to evolve in different granularity levels, by considering a loose-dependency between the services and the elements used to describe them. We describe how the version framework and compatibility assessment method were prototyped in a SOA environment, and the lessons learned.

We focus on *backward compatibility*, which is concerned with how changes to the service interface affect existing clients [1], [2], [5], [7], [9], [13]. In the rest of the paper, we shall use *compatibility* as synonym for backward compatibility. A service version is defined to be backward compatible with a previous one if it: a) delivers at least the same functionality; b) possibly relaxes constraints on the input expected while delivering the same results; and c) generates outputs that can be consumed by existing clients

The remaining of this paper is structured as follows. Section 2 describes the versioning framework proposed to assess service compatibility. Section 3 describes the rules and the algorithm to check compatibility between service versions, using an object-oriented service description as an illustration. Section 4 describes our implementation of a compatibility-aware SOA, and discusses performance and benefits. Related work is described in Section 5, and conclusions and future work, in Section 6.

2. Service versioning

As pointed out by Frank et al. [8], versioning is an overloaded term in the service context. From the client point of view, a service version refers to the "contract" established by the interface of the service, and consequently, to the functionality a service delivers and the data types which it exposes within the interface. From a service provider point of view, however, version refers to a particular implementation of the service, and how the service implementation evolves over time, in addition to any changes in the interface definitions. We refer to these two aspects as the *service model* and *service implementation*, respectively. Because we are concerned about compatibility between services and clients, this paper concentrates on the evolution of the service model, while assuming that service implementation issues are handled using regular source versioning systems [3].

We assume that the service is represented within the SOA by a *model schema*, which defines the external representation of the service as a set of versioned abstractions and relationships between those abstractions. We refer to these abstractions collectively as *types.* The service model defines the collection of types exposed by the service within the SOA. When a service implementation evolves, changes are reflected in the corresponding service model with the update, addition or removal of types that represent updated, added or removed service functionality.

The service model is important from the view of both the service implementation and the client. For the service implementation, it defines the types that are instantiated, validated and controlled by the implementation; for the client, it describes the functionality provided by the service and the message syntax and semantics necessary to use the service. Awareness of the service model is also important for the SOA itself to enable it to validate or redirect client requests to the appropriate service implementation, and handle service evolution issues accordingly.

2.1. Versioning framework

Integration is a key value for the success of SOA, and therefore, it is desirable that types are reused (or shared) as much as possible across services. This is particularly important in the context of service composition, where message content is transparently forwarded in the context of composed services. A type describes part of the service functionality or properties of exchanged data. Thus, flexibility in service description and integration is achieved by considering a loose relationship between types and the service models they help describe. To achieve such independence, we assume that types can evolve independently from service models.

Figure 1 depicts the proposed version framework for addressing compatibility. The framework defines two kinds of versioned units (*VersionUnit*): *TypeVersion,* and *ModelVersion*. Figure 2 shows examples of constraints defined over this framework using OCL. For instance, the first invariant asserts the sameness criterion (*name*) and unique identifier (composition of *name* and *version*) of any *VersionableItem* [3]. *VersionUnit.version* is typically a string composed of decimal integers separated by periods. However, unlike other versioning systems we assume no specific semantics about compatibility between versions based on their version numbers.

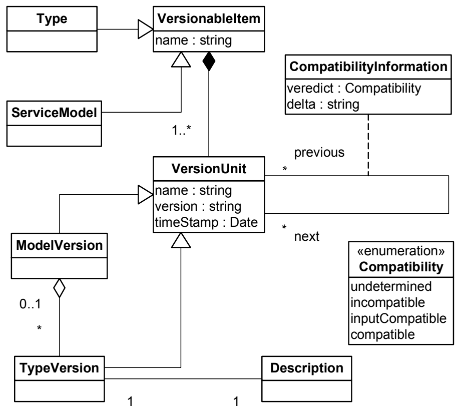


Figure 1. Versioning framework

```
"All versions related to a VersionableItem share the
same name and have different version numbers"
context VersionableItem inv: self.VersionUnit->
forAll(v | self.name = v.name) and forAll(v1, v2 | v1
<> v2 implies v1.version <> v2.version))

"Versions of a Type are TypeVersion"
context Type inv: self.VersionUnit-
>forAll(oclIsKindOf(TypeVersion))

"Versions of a Service Model are ModelVersions"
context Service inv: self.VersionUnit->
forAll(oclIsKindOf(ModelVersion))

"A model version cannot be described in terms of two
versions of a same type"
context ModelVersion inv: self.TypeVersion-
>forAll(t1, t2 | t1 <> t2 implies t1.name <> t2.name)
```

Figure 2. OCL constraint excerpts

Any *TypeVersion* has a corresponding *Description* that provides the definition for that type. The *Description* may assume any service description paradigm, such as excerpts of WSDL and XSD descriptions, or classes and associations within an object-oriented (OO) paradigm. This approach of relating versioned units and abstract elements is similar to the one proposed by Murta et al. [12] in the context of configuration management for UML case tools.

A *ModelVersion* is composed of a set of *TypeVersion* instances, at most one per type. We say a model version *contains* a type if it is related to one of its versions.

A new *VersionUnit* may be compared with any previous one to determine their compatibility. The precedence between versions can be determined by different criteria, such as version numbers or timestamps. We refer to this precedence between versions as $v_1 < v_2$, where v_1 is an older version and v_2 is a newer version. Due to the focus on backward compatibility, compatibility is evaluated considering the *directed* delta between the new version and previous ones, i.e. by using the sequence of change operations $op_1...op_m$ required to transform v_1 into v_2. In the case of *TypeVersion*, a delta defines the operations required to change a previous description into the newer one, by adding, removing, or changing properties or operations defined within that description. In the case of *ModelVersion*, a delta refers to the addition of a *Type* (i.e. the previous model version did not contain a version of that type), removal of a *Type* (i.e. the new model does not contain a type contained by the previous model), or update (i.e. replacement of a version by another one of the same type, typically, a more recent one).

Compatibility is determined by analyzing the delta between the two versions. There are different criteria for determining compatibility of *TypeVersion* and *ModelVersion*, where the latter depends on the former. These are addressed in Section 3, using OO descriptions to describe the service model as an illustration.

3. Compatibility assessment

We propose a two-phase compatibility assessment approach for service descriptions, based on the premise that types are shared across service descriptions, and thus they must evolve independently of the service model. In the assessment of type compatibility, not all contextual information for a final decision is available: the versions of the types to which a given type is related are not known in advance, nor are the specific dependencies with other types within a model. Model-based compatibility assessment complements type-level assessment, by putting type versions in the context of the model version and thus defines the scope of their relationships.

We illustrate the approach using compatibility rules defined over OO descriptions of services. This choice is justified by the higher level of independence allowed between types in this paradigm. We note that the approach can also be applied to other types of service descriptions, notably combinations of WSDL and XSD. Indeed, any XML complex type built over simple elements can be regarded as a class, and relationships that form more complex XML structures as associations between those classes. The compatibility rules proposed are similar in nature to the ones discussed in the context of web services (e.g. [1], [2], [5], [7]).

3.1. Type compatibility assessment

Compatible changes that can be applied to types are summarized in Table 1. We assume that types are described by classes, and attributes and operations defined within those classes. Relationships between classes are declared using associations. It should be noted that changes on types used as *parameters* within operations must be viewed from two perspectives, as represented by their role as input or output parameter. Thus type compatibility is in some cases sensitive to the context within which the type is used and depends on the overall schema. The column labeled *Output Restriction* in Table 1 highlights the cases that are backward compatible only if they do not affect an output of some operation (perhaps defined in some other type). Thus from a client point of view, the results of the operation must conform to a type that guarantees all expected information, functionality and constraints. From a service point of view, constraints can be relaxed as long as the service implementation guarantees it can still provide the same functionality.

It should also be noted that compatibility is not restricted to the syntactical properties of types. Semantics can be conveyed by informal notes or formal constraints (e.g. as represented by UML metaclasses *Comment* and *Constraints* [14]). Currently, we treat these as mere textual elements, such that any change on these elements is considered incompatible.

Given two *TypeVersion* t, v, where $t < v$ and delta(t, v) is not empty, the compatibility of t with regard to v is labeled as:

- *Incompatible*: if at least one operation o in delta(t,v) is not listed in Table 1;
- *InputCompatible*: if all operations o in delta(t,v) are listed in Table 1, and at least one of them is output restricted.
- *Compatible*: if all operations o in delta(t,v) are listed in Table 1, and none of them is output restricted.

Note that if $t > v$, the compatibility of v with regard to t is *undetermined*, because backward compatibility is not a symmetric relationship.

3.2. Model compatibility assessment

Model versions are sets of versioned types, and their compatibility is assessed in terms of update, addition and removal of types as defined in Section 2.1. The removal of types is in all cases labeled *incompatible*. However, the addition and update of types usually require contextual information about how the underlying type versions are related in the model. In addition to type version compatibility, the following relationships between type versions are of relevance for model compatibility assessment: sub-classing, dependencies established by strongly typed arguments or attributes, and participation in associations. It should be noted that different descriptions related to these sets of type versions result in different model schemas. Compatible changes are listed in Table 2, and all others are regarded as incompatible. Given two

ModelVersion x and y, where $x < y$ and delta(x, y) is not empty, the compatibility of y with regard to x is labeled as:

- *Undetermined:* if at least one operation o in delta(x, y) replaces a version v of a type t by a version v' of the same type, where $v' < v$;
- *Incompatible*: if there is at least one operation in o in delta(x, y) that is not listed Table 2, and this operation does not qualify the model's compatibility as *Undetermined*;
- *Compatible*: if all operations o in delta(t, v) are listed in Table 2.

Because the assessment described in Table 1 is straightforward, the algorithm for determining type compatibility is omitted. Figure 3 presents the pseudo-code for processing the model version compatibility. The algorithm is based on the data structure depicted in. Figure 4. Methods are used as explained in the text below, or their semantics can be inferred from Figure 4. The algorithm assumes that all type versions are known as part of the registration process of a new release of the model, together with the corresponding compatibility assessment considering relevant earlier type versions, as discussed in Section 2.

Table 1. Type level backward compatible changes

Update Operation	Type Element	Description	Output restriction
add	Operation	Add a new operation	
change signature: input parameter	Operation	Change lower bound cardinality from mandatory to optional	
change signature: input parameter	Operation	Change parameter class to superclass (immediate or not)	
add	Attribute	Add new attribute where lower bound cardinality is optional	
change	Attribute, association participant	Change cardinality lower bound from mandatory to optional	yes
change	Attribute, association participant	Change cardinality upper bound from (1) to (*)	yes
change	Attribute, association participant	Change referenced class to a superclass (immediate or not)	yes

Table 2. Schema level backward compatible changes

Operation	Model Element	Description	Output restriction
Add	Class	Add a type t in new-model	
Update	Class, Association	Replace version v of type t in previous-model by a *compatible* version v' of t in new-model	
Update	Class, Association	Replace version v of type t in previous-model by an *inputCompatible* version v' of t in new-model	yes
Add	Association	Add type t in new-model, such that if a participant class in the description of t refers to a type u contained previous-schema, the lower-bound cardinality constraint must be optional	

```
verifyCompatibility (pModelVersion: ModelVersion, nModelVersion: ModelVersion): CompatibilityResult;
begin
CompatibilityResult result;
ServiceSchema pSchema, nSchema;
SchemaComponent pcomp, ncomp;
1. pSchema := buildSchema(pModelVersion);
2. nSchema := buildSchema(nModelVersion);
3. result.setCompatibility(COMPATIBLE);
4. for each ncomp in nSchema.components()
5. do begin
6.    pcomp := findCorrespondentComponent(pSchema.components(), ncomp.myType());
7.    if pcomp = null
8.    then processAddedType(ncomp, result);
9.    else begin
10.       compatibility := myType().getTypeCompatibility(pComp.myType()).compatibility;
11.       if compatibility = INPUT-COMPATIBLE and not pcomp.isUsedAsOutput()
12.       then compatibility = COMPATIBLE
13.       else compatibility = INCOMPATIBLE;
14.       pcomp.setProcessed(true);
15.       end;
16.   end;
17. for each pcomp in pSchema.components()
18. do if not pcomp.isProcessed()
19.    then processRemovedType(ncomp, result);
20. return(result);
end;
```

Figure 3. Model compatibility comparison algorithm

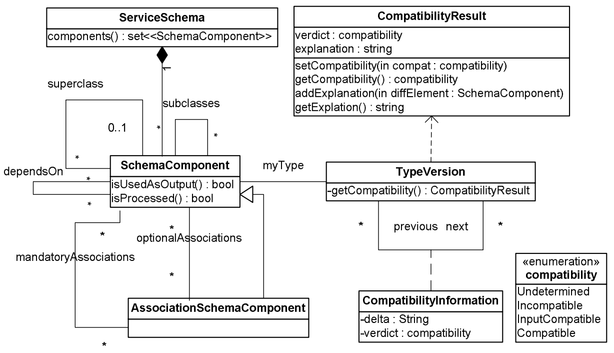


Figure 4. Compatibility checking data structure

The first step in the algorithm is to build the schema structure depicted in Figure 4 for each of the compared model versions (lines 1 and 2). To create this schema representation, `buildSchema` considers all descriptions associated with the type versions included into each model, and creates schema components that are related to each other through the relationships required for investigating compatibility, i.e., superclass, subclass, associations in which they participate, dependencies as operation parameters and strongly typed attributes. In addition, it evaluates if the type is used to strongly type an output argument anywhere in the schema. Lines 4 to 10 determine compatibility of the schemas by comparing each component of the new schema with the corresponding one of the previous schema. Type names are used to establish such a correspondence between the versions (line 6). If the type was not contained in the previous model, then it is verified if its inclusion does not break compatibility according to addition rules in Table 2 (line 8). If the type is replaced by a new version, then its compatibility needs to be resolved (lines 10 to 14). As components of the previous schema are compared to the corresponding ones of the new schema, they are marked as processed (line 14). Finally, we search for removal of types, and therefore we search for all non-processed types in the previous schema (lines 17 to 19). Notice that the algorithm processes all types in both models, such that it produces not only a final verdict on compatibility between the models, but also produces a complete explanation of the delta.

4. Case study for compatibility assessment

We are testing our solution within the Shared Services Platform (SSP), which melds SOA and model-driven architecture for describing IT services [15]. All services in the SSP are hosted at a *Service Access Points* (SAP) and interact through service models. The SAP is a proxy service that provides, among other things, message routing and validation capabilities. SSP adopts the OO paradigm for service description, and it is implemented using Java, OSGi technology [17], and Oracle Berkeley DB for persistence support [16]. The SSP provides an ideal test case for understanding how version compatibility awareness can be leveraged into the SOA. Figure 5 depicts the functional components developed for version awareness in the SSP, which are briefly described below.

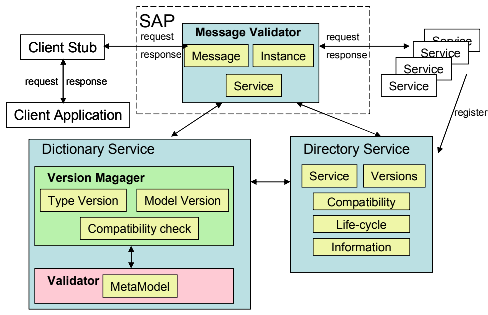


Figure 5. Functional components within the SSP for compatibility assessment

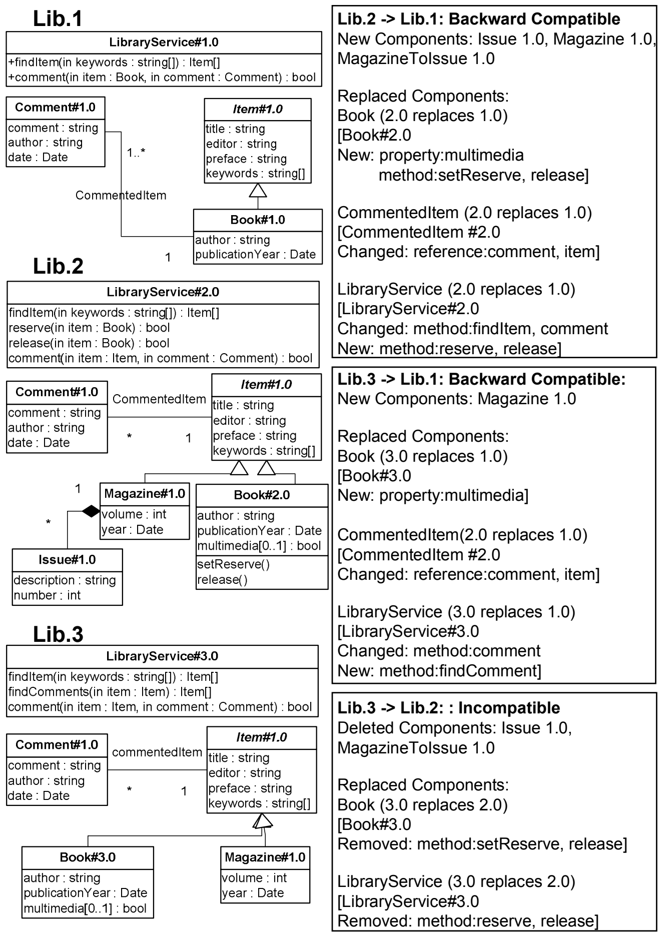
Dictionary Service: The Dictionary is the authoritative repository for all types and models in SSP. These are represented according to the versioning framework described in Section 2.1. Version numbers must be explicitly provided by type authors. The Dictionary checks compatibility of all type versions and model versions when they are inserted by extracting their directed delta and assessing the compatibility of their differences. An example of compatibility assessment is provided in Section 4.1. In addition, the dictionary ensures that all descriptions conform to the meta-model adopted for representing types and models. Currently, the SSP implementation assumes that models are declared using the OO CIM meta-model [4].

Directory Service: The Directory provides mechanisms for publishing and finding service descriptions, and is similar to a registry service. The distinction is that services are versioned, and that in addition to providing the normal discovery functions, the directory also maintains compatibility information among different versions of the service. For this purpose, it interacts with the Dictionary Service. In addition, the directory maintains information about service release lifecycle according to three perspectives: a) availability (*available, deprecated, unavailable*); b) stability (*stable, experimental, unknown*) and c) alias (*new, old, current*). The alias is used to convey an external representation of the recency of the service release.

Message Validator: The message validator is implemented within the SAP and guarantees consistency of messages exchanged in the SOA environment. Using the description of the service model and versioning information, it can guarantee that the service can indeed interpret the request; and that the data in message is compatible with the service's model. In addition, it interacts with the Directory to find out the availability of the service version.

4.1. A compatibility assessment example

Figure 6 exemplifies the output of the compatibility assessment algorithm in our Dictionary implementation. In the picture, service descriptions are depicted using a UML representation. Three different model versions are presented for service *Lib*, namely *Lib.1*, *Lib.2* and *Lib.3*. The respective service models are based on types such as *LibraryService*, *Item*, *Book*, *Comment*, etc, which in turn are versioned.



Lib.2 -> Lib.1: Backward Compatible
New Components: Issue 1.0, Magazine 1.0, MagazineToIssue 1.0

Replaced Components:
Book (2.0 replaces 1.0)
[Book#2.0
New: property:multimedia
method:setReserve, release]

CommentedItem (2.0 replaces 1.0)
[CommentedItem #2.0
Changed: reference:comment, item]

LibraryService (2.0 replaces 1.0)
[LibraryService#2.0
Changed: method:findItem, comment
New: method:reserve, release]

Lib.3 -> Lib.1: Backward Compatible:
New Components: Magazine 1.0

Replaced Components:
Book (3.0 replaces 1.0)
[Book#3.0
New: property:multimedia]

CommentedItem(2.0 replaces 1.0)
[CommentedItem #2.0
Changed: reference:comment, item]

LibraryService (3.0 replaces 1.0)
[LibraryService#3.0
Changed: method:comment
New: method:findComment]

Lib.3 -> Lib.2: : Incompatible
Deleted Components: Issue 1.0, MagazineToIssue 1.0

Replaced Components:
Book (3.0 replaces 2.0)
[Book#3.0
Removed: method:setReserve, release]

LibraryService (3.0 replaces 2.0)
[LibraryService#3.0
Removed: method:reserve, release]

Figure 6. Compatibility assessment example

In *Lib.1*, the service provides operations to find any library item based on keywords, and to provide comments on a book. In *Lib.2*, it becomes possible to provide comments on any library *Item*, and therefore the input argument of operation *comment* is generalized. The relationship *CommentedItem* that relates *Comment* and *Book* is also changed to support that (role generalization, minimum cardinality). Another incremental feature is the ability to *reserve* books, as well as the explicit control of *Magazines* and their respective *Issues*. In *Lib.3*, the service provider decides that these new features do not pay off (e.g. they not used or they are too expensive to maintain) so operations *reserve* and *release* are removed (from both *LibraryService* and *Book*), and so is explicit *Issue* control. Considering these changes, the Dictionary service registers that both *Lib.3* and *Lib.2* are backward compatible to *Lib.1*, but that *Lib.3* is incompatible with *Lib.2*.

4.2. Performance Experiments

Experiment set-up. The SOA environment for our experiment was located in a single machine (Pentium 2.8GHz, 2 GB RAM, Windows XP). The SOA contained the Directory Service and the Dictionary Service. We considered for the experiment the registering of a fire simulation service (FDS), of which the description is composed of 52 types (classes and associations).

Experiment Description. We compared the cost of registering a new version of the FDS in the Directory Service with compatibility checking and without. We generated up to 20 model versions for FDS. Changes were introduced with regard to the immediate previous version at both type and model levels. We tested 3 distinct change scenarios: changes on 10%, 30% and 50% of the types (respectively 5, 15 and 26 types). Changes were performed always in the same set of types, in which compatible or incompatible updates were randomly introduced. We then measured the time necessary to register a new service version with and without compatibility checking considering 4, 9, 14 and 19 prior service versions (and consequently, the same amount of versions for the types changed). Each execution was repeated 5 times, and the average results are displayed in Table 3, together with the corresponding standard deviation in parenthesis. Time is measured in milliseconds. Rows are labeled considering a combination of the percentage of change introduced, and register operation without compatibility checking (OFF) and with compatibility checking (ON). The columns represent the number of previous releases that the new version needs to be compared with. The graph shown in Figure 7 shows the overhead introduced in the service registering operation by the insertion of the compatibility checking, considering the number of previous releases and the percentage of changes applied.

Discussion. As it can be seen in Table 3, the absolute time for registering a new service version with compatibility checking is not significant. In the worst case examined, it amounts to 29 seconds. However, it does introduce a significant overhead to the registering operation: from 91 times in the most favorable scenario, to 233 times in the worst case examined. These results were expected, due to the combinatorial nature of the algorithm. Indeed, a new model version is compared with all previous ones, which implies in turn comparing the new type versions with all previously existing types contained in those model versions. Nevertheless, we believe that the chosen persistence technology accounts for some of the performance issues. It can be noticed an inverse relationship between % of changes and time required to register a new model version without compatibility checking. Recall that version numbers are provided by service authors, and therefore, if a type version is already registered, current implementation further checks if the type description provided matches the registered one.

Table 3. Time to register a new service version with and without compatibility checking

	Previous Service Releases			
Percentage of Changes	**4**	**9**	**14**	**19**
10% Comp. checking. OFF	172 (0)	188 (0)	187 (0)	172 (0)
10% Comp. checking ON	15580 (202)	18721 (153)	22085 (275)	25205 (191)
30% Comp. checking OFF	171 (0)	156 (0)	140 (0)	157 (0)
30% Comp. checking ON	16886 (163)	20267 (171)	23609 (132)	26965 (148)
50% Comp. checking OFF	143 (0)	140 (0)	125 (0)	125 (0)
50% Comp. checking ON	17941 (225)	21415 (202)	25381 (198)	29079 225)

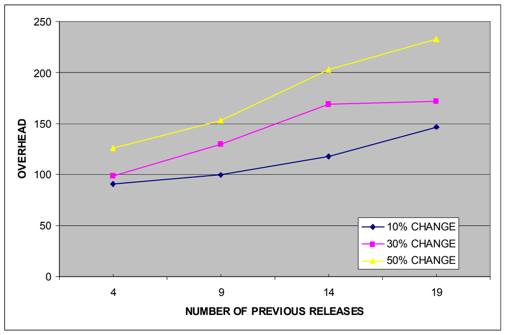


Figure 7. Overhead introduced in the registering operation

4.3. Benefits of compatibility-aware SOA

By leveraging version compatibility into the SSP, the following benefits are observed:

Service Updates: During development, service descriptions are changed to introduce new functionality. Often, developers wish to offer new functionality without compromising existing functionality. As service descriptions become complex, it is easy to mistakenly introduce incompatible changes. To help the developer avoid such mistakes, the Dictionary may be used to evaluate service compatibility with regard to previous versions, and correct the service description in case of errors before the service is made available to clients.

Service Discovery: Clients discover services and their models by performing queries on the Directory and Dictionary services. By introducing versions in to these services, clients can request specific versions of a service during lookup, and are aware of the version of all services returned. They can also request for compatibility information. Thus, as client applications are developed and deployed, proper versioning requirements can be put in to the implementation guaranteeing that incompatibilities will not occur during run-time.

Service Binding: The run-time corollary to discovery is binding. This is the time when service instances are selected for use by a particular client instance. It is possible that a new (perhaps incompatible) version of the service is released between the time the client is developed and the time it is put into operation. To prevent run-time failures due to change, a client may use the information collected during discovery to make sure that a bound service is compatible with its requirements. During binding, the client can present the service interface definition used during development, and be assured that the present service interface is compatible. If so, it can use the service successfully, and if not, errors can be detected early rather than waiting for failures to occur when the service interactions happen.

Message Validation: The SOA run-time or middleware systems are responsible for transporting messages from a client to a service. Typically, run-time systems do simple validation of the structure of a message such as matching all begin and end tags in an XML document. However, it is usually not possible for the run-time to guarantee that messages contain the type of information expected by a service. Within the SSP, the SAP handles a number of validations, including the ability to handle a request for validity against the expected service description, as well as compatibility with it. In particular, if message content is forwarded to upgraded services as part of some service composition, compatibility checks ensure that the service receives only messages that it expects or backward compatible messages. Notice that individual type compatibility is very important for this purpose.

5. Related work

Versioning is extensively addressed within the context of software development, but the compatibility of the deltas extracted from service descriptions and their effect in the whole configuration is not addressed in prior literature [3]. Most existing work in web service versioning (e.g. [1], [2], [5], [10], [13]) discusses design patterns and best practices within the context of relating service descriptions and implementation. They also provide a framework to evaluate backward compatibility. In a more restricted way, forward and backward compatibility of XML documents updates is discussed by Moro et al. [11] with the goal of helping designers to understand the resilience to changes of XML schemas and respective queries.

Registry solutions (e.g. UDDI v3.0.2, Systinet, [7]) provide mechanisms for registering and documenting releases of the same service (e.g. for discovery or governance purposes), but they make assumptions about compatibility (e.g. convention on version numbers, only backward compatible releases). Functional components for embedding versioning into the SOA are discussed by Fang et al. [6] and Frank et al. [8], notably with the proposal of service redirection based on compatibility. However, no support is provided for assessing such compatibility other than guidelines to be followed by service developers. A design technique is proposed by Kaminski et al. [9], in which a chain of adaptors deal with the (in)compatibility between the current and the previous releases of a service. However, they do not discuss how to assess the differences as an input to the adaptor design.

6. Conclusions and future work

In this paper, we described a framework for automatically assessing backward compatibility between two revisions of a service as it undergoes evolution as a result of lifecycle management. This is important because within SOA, the development lifecycles of services and clients are de-coupled, and it is important that services continue to support older clients. Compatibility assessment is error-prone if done manually, especially for service descriptions that are complex.

The distinctive features of our approach are: a) It implements automatic detection of type and service schema compatibility, which in related work is limited to a set of (loose) guidelines; b) It allows loose coupling between type versions and their usage to define service descriptions. The latter is a more flexible and comprehensive solution for service description evaluation in the presence of integration issues for services composition. We implemented our framework within a SOA case study, and shown benefits for clients and providers of exploring automatically compatibility information. Early performance tests have shown satisfactory results, which can be improved by the introduction of optimization techniques to reduce the combinatorial effect. We are currently undertaking is the insertion of version derivation meta-information (e.g. type 2.0 modifies type 1.0), such that type versions are organized in a tree [3], where the edges represent derivation relationship labeled with compatibility verdict. In this way, we can infer about the compatibility of versions in the same derivation path, and significantly reduce the number of comparison within type versions.

We are currently developing a more comprehensive study of semantic compatibility, using OCL constraints. As future work, we plan to generalize compatibility rules to other services description approaches, notably semantic ones such as WSDL-S, OWL-S or SML. We will add subscription/notification mechanisms in the Directory such that clients can become aware of the lifecycle stage of the versions they use, as well as the availability of newer versions. We will consider performing automated re-direction of compatible messages, so that they can reach a compatible service in case the originally intended service version becomes unavailable. Finally, we plan automated creation of translation adapters, to minimize human engagement in the upgrade.

References

[1] Bachmann, R. “Challenges of Web Service Change Management,” https://www.sdn.sap.com/irj/servlet/prt/portal/prtroot/docs/library/uuid/4e1d4d29-0801-0010-159b-f8d51a04bbbd

[2] Brown, K.; Michael, E. “Best practices for web services versioning, www.ibm.com/developerworks/webservices/library/ws-version

[3] Conradi, R. and Westfechtel, B. Version models for software configuration management”. ACM Computing Surveys, vol. 30, no. 2, pp.232-282, 1998.

[4] DMTF. “Common Information Model (CIM) Specification”, version 2.2. 1999,. http://www.dmtf.org/standards/cim/cim_spec_v22.

[5] Endrei, M. et al. “Moving forward with web services backward compatibility”. http://www-128.ibm.com/developerworks/java/library/ws-soa-backcomp/

[6] Fang, R. et al. “A version-aware approach for web service client application”. 10th IFIP/IEEE International Symposium on on Integrated Network Management, 2007. pp. 401 – 409.

[7] Fang, R. et al. “A version-aware approach for web service directory”. ICWS 2007. pp. 406 – 413

[8] Frank, D. et al. "An Approach to Hosting Versioned Web Services". IEEE SCC 2007: pp. 76-82

[9] Kaminski, P., Litoiu, M., Müller, H. A design technique for evolving web services. CASCON 2006. pp. 303-317.

[10] Lublinsky, B. “Versioning in SOA”. Microsoft Architect Journal. msdn2.microsoft.com/en-us/arcjournal/bb491124.aspx

[11] Moro, M., Malaika, S., Lim, L. Preserving XML queries during schema evolution. WWW 2007. pp. 1341-1342

[12] Murta, L. et al. Odyssey-SCM: An integrated software configuration management infrastructure for UML models. Sci. Comput. Program. Vol. 65, No. 3. pp. 249-274, 2007.

[13] Narayan, A. and Singh, I.. Designing and versioning compatible Web services. http://www.ibm.com/developerworks/websphere/library/techarticles/0705_narayan/0705_narayan.html

[14] OMG, OMG Unified Modeling Language (OMG UML) - Infrastructure (V2.1.2). http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF

[15] Singhal, S.; Machiraju, V.; Pruyne, J. “Picasso: A Services Oriented Architecture for Model based Automation,” HP Labs Technical Report HPL-TR-2007-50R1.

[16] Oracle Berkley DB Java Edition. http://www.oracle.com/technology/products/berkeley-db/je/index.html

[17] OSGi Alliance http://www.osgi.org/Main/HomePage.