Ontology-Based Compatibility Checking for Web Service Configuration Management

Qianhui Liang[1] and Michael N. Huhns[2]

[1]School of Information Systems,
Singapore Management University, Singapore
[2]Department of Computer Science and Engineering,
University of South Carolina, USA
althealiang@smu.edu.sg,huhns@sc.edu

Abstract. Service-oriented systems are constructed using Web services as first-class programmable units and subsystems and there have been many successful applications of such systems. However, there is a major unresolved problem with the software development and subsequent management of these applications and systems. Web service interfaces and implementations may be developed and changed autonomously, which makes traditional configuration management practices inadequate for Web services. Checking the compatibility of these programmable units turns out to be a difficult task. In this paper, we present a technique for checking compatibility of Web service interfaces and implementations based on categorizing domain ontology instances of service description documents. This technique is capable of both assessing the compatibility and identifying incompatibility factors of service interfaces and implementations. The design details of a system model for Web service compatibility checking and the key operator for evaluating compatibility within the model are discussed. We present simulation experiments and analyze the results to show the effectiveness and performance variations of our technique with different data source patterns.

Keywords: Web services compatibility, ontology, Web services configuration management.

1 Introduction

As service engineering takes hold in large-scale intra- and inter-enterprise service-oriented computing (SOC) settings, how to manage service configurations (or differences) becomes a key enabling task for correct and effective use of dynamic services. Service components, e.g., Web services, resemble traditional independent software systems in being autonomous. However, they have an important advantage over traditional systems: they are network discoverable and accessible via standardized protocols. Comparing service compatibility is the basis of

A. Bouguettaya, I. Krueger, and T. Margaria (Eds.): ICSOC 2008, LNCS 5364, pp. 407–421, 2008.

configuration management for Web services, as it is the key to successful engineering of the third generation of Web services, such as (1) forming a consensus about each other's behavioral differences and similarities, and (2) robust service composition by self-organizing similar services into teams [1].

Configuration management practices in traditional software engineering [2,3,5][13,14,15] are insufficient for Web services. One major reason is that the autonomous development and deployment of services does not constrain the service engineering and management activities within a tightly-coupled and closely administered environment as is typically expected for traditional local components or controlled distributed components. For example, in a service-oriented environment, two very different service providers may independently evolve the same service interface. They may also develop and change service implementations according to the evolved version of the interface. Given the fact that changes to Web services are not performed by people that are bound by the same organizational constraints, it is important to discover and maintain a consistent mechanism to track the compatibility or incompatibility of the services. Such a mechanism will have a major impact on the reuse of services and on maintaining a consistent understanding of the evolutions made by different parties in a holistic way.

In this paper, we first present a conceptual system model for service objects to facilitate compatibility checking in a service-oriented environment. We then present a technique that conforms to this model to be used for the compatibility checking of Web service interfaces and implementations, which are possibly developed by different teams or completely independent providers. The technique enables consistent Web service configuration management in an enterprise or open service-oriented environment. This technique can be used to (1) retrieve Web service interfaces or implementations that are compatible with a given service object, and (2) identify distinct incompatibility factors of service interfaces and implementations. The technique is based on a domain ontology instance categorization tool, which categorizes terms from service description documents. The technique defines several semantic rules, which are designed to be applied to the term categorization results for assessing compatibility and identifying incompatibility factors of service objects. We present the experiments and results, and demonstrate the effectiveness of our approach on checking compatibilities of services.

The remainder of the paper is organized as follows: Section 2 is related work. Section 3 describes the idea of compatibility checking of Web services and a conceptual system model for organizing service objects and checking compatibility of service objects within this model. Section 4 discusses the technical details of an ontology instance categorization tool, presents the scheme of using ontology categorization for service compatibility checking, and further elaborates how to use the semantic results to distinguish various incompatibility factors. Section 5 illustrates the experiments and provides an analysis of the experimental results, while Section 6 concludes the paper.

2 Related Work

Automatic service discovery and selection is a key aspect for composing Web services dynamically in SOC. Current approaches to automating discovery and selection make use of only structural and functional aspects of the Web services. We believe that behavioral selection of Web services should be used to provide more precise results. Service behavior is difficult to specify prior to service execution and instead is better described based on experience with the service execution. In earlier work, we presented an approach to service selection and maintenance-inspired by agile software development techniques-that is based on behavioral queries specified as test cases. Behavior is evaluated through the analysis of execution values of functional and non-functional parameters. The tests can also be used to assess performance and reliability. Therefore, in addition to behavioral selection, our framework allowed for real-time evaluation of non-functional quality-of-service parameters, scalability, and dynamism [4]. Our work reported herein focuses on the compatibility issues of service evolution with multiple providers.

Ontology studies for semantic Web services can be categorized as ontology matching, ontology mapping, and ontology merging. Techniques and algorithms have been proposed to match concepts among heterogeneous ontologies in a tree-like structure or a graph-like structure [9,10,11]. A majority of the matching techniques proposed are schema-based and discover similarities from linguistic or (and) structural characterizations. Platforms and frameworks that provide an integrated environment to facilitate easy use of these matching techniques are also reported. Our compatibility checking system relies on a probabilistic ontology categorization technique. Similar to GLUE [12], the categorization technique is also instance based. In contrast to GLUE, the aim of our ontology categorization technique is to provide a strong evidence for Web service compatibility instead of producing a merged ontology.

As the adoption rate of service-oriented computing and Web services continues to grow, topics pertinent to service engineering are attracting increasing interest. The materialized form of services is software products, which are required to go through a number of general engineering activities. The bottom line of service engineering, therefore, is a process similar to the Software Development Life Cycle (SDLC), which consists of design, build, test, and maintainance. One important aspect of the process is how to record the changes so that consistency and compatibility is guaranteed for interactions between service users and providers. A few industry papers present best practices in Web service versioning [6,7]. Frank et al. [8] also present a proxy-based Web service hosting environment with routing points to handle requests of different versions. They have separate versioning for service interfaces and service implementations. The aim of our work on compatibility checking is to address the most basic issue of Web service versioning and to lay the foundation for research on Web service-specific versioning systems.

All the research issues studied in the above related work can be attributed to one fact, i.e., unlike traditional software, services are published and shared across the Web. As a result, the changes and evolution of service interfaces and implementations are not controlled by a single authority. As more services that serve the same purpose become available, improved versions of the services will be an increasingly complex problem to solve. For example, two important and open issues to be addressed for Web service versioning is how compatibility of services can be determined in the presence of heterogeneous descriptions and how versions of services can be constructed based on compatibility.

3 Web Service Compatibility and Versioning

In this section, we introduce our compatibility model of Web service interfaces and implementations.

3.1 Web Service Compatibility Model

To study the compatibility of service interfaces and service implementations, we propose a schema that models the following two categories of information, each of which is represented as a class in the model:

- The classes of objects useful for compatibility checking
- The families that collect related objects.

Object classes include a selection of elements defined in WSDL and UDDI, i.e., `businessService`, `operation`, `tModel`, and `categoryBag`. The classes are defined in the schema in Listing 1. Objects of these classes are referred to as service objects. The first three classes all derive from a class of Versioned objects. We treat compatibility of service interfaces and service implementations as two separate but related issues. We also consider versioning of service operations. Therefore, we use the above service objects in our model to differentiate levels of compatibility checking that may contribute to Web service configuration management.

Attributes of a class can be single-valued, such as release and time of the Versioned class. Attributes can also be set-valued, such as `iTypes` of `TModel` class. Compatible objects are linked together. The previous object and the next object in the same link are pointed to by attributes of `nextCompa` and `prevCompa` of a particular object. The number of trailing objects is recorded in tail.

The class of `CheckingSystem` represents the entire repository established for the compatibility checking purpose, which contains all objects of all classes. The attribute of `portTypeRoots`, `tModelRoots`, `businessServiceRoots` are the roots of the linked compatible objects. It also has three Check constraints in `CheckingSystem`. The first Check constraint specifies that each `TModel` object must be categorized with one `CategoryBag` object. The second Check constraint specifies that each `TModel` must have at least one `PortType` object defined in it. The third Check constraint specifies that each object of `BusinessService` object must implement at least one `TModel` object.

Listing 1

```
{
Versioned: class (
   release: Integer,
   build: Integer,
   time: TimeStamp)
NameOwner: class (
   name: String,
   owner: String)
Operation:
   subclass of Versioned, NameOwner (
   operation: XML,
   nextCompa: InterfaceOperationType,//defined in WSDL20
   prevCompa: InterfaceOperationType,
   tail: int)
TModel:
   subclass of Versioned, NameOwner (
   tModel: XML,
   cBags: set of CategrotyBag,
   iTypes: set of InterfaceType,//defined in WSDL20
   nextCompa: TModel,//defined in UDDI3
   prevCompa: TModel,
   tail: int)
CategoryBag:
   subclass of NameOwner (
   categoryBag: XML,
   tModel: TModel)
BusinessService:
   Subclass of Versioned, NameOwner (
   bindingTModels: set of TModel,
   nextCompa: BusinessService,//defined in UDDI3
   prevCompa: BusinessService,
   tail: int)
CheckingSystem: class (
   operations: set of Operation,
   tModels: set of TModel,
   businessServices: set of BusinessService
   categoryBags: set of CategoryBag
   check
(for-all tm in tModels)
(there-exists b in categoryBags)
   check
(for-all tm in tModels)
(there-exists i in iTypes)
   check
(for-all bs in businessServices)
(there-exists tm in tModels))
}
```

3.2 System Model

We now describe the structure of our compatibility-checking system. Listing 2 defines the system model of the compatibility checking system. `checkingSystem` is an instance of class `CheckingSystem`. It contains version families composed of objects of class `Operation`, `TModel`, and `BusinessService`. We have used the concept of version family when designing our compatibility-checking system. Our purpose is to categorize the object space into a number of compatible groups. A version family is a group of objects of the same class selected based on certain properties [18]. These compatible objects (belonging to the version) are linked together.

The system must be capable of refining the compatibility classification by selection of version families. We use version selection rules for this purpose. These rules operate on version families and select one or more members from the families. In particular, the selection rules of compatibility checking return a set selected from the compatible objects. These objects may or may not belong to the same family, depending on the requirement of the checking task. Several examples are given in Listing 2.

The `select` operation in Listing 2 outputs the latest `tModel` object in a `tModel` version family. `selectCompatibles` operation outputs all `tModel` objects in a version family that are compatible with a given tModel object. The `incompatiblewith` operator outputs 0 if the first operand is compatible with the second operand. Otherwise, it outputs some positive number that identifies incompatibility causes if the first operand is incompatible with the second. The technique to implement the `incompatiblewith` operator is discussed in Section 4. `selectCompatibleRoot` finds the `tModel` root that has the largest number of objects in its tail.

Listing 2

```
checkingSystem: system model from CheckingSystem (
   operations = {"Root1_operations", "Root2_operations, }
   tModels = { "Root1_tModels", "Root2_tModels",  }
   businessServices = {"Root1_businessServices", "Root2_businessServices",  }
   categoryBags = {"Finance", })
select tModel from f: family of TModel
  in checkingSystem.tModels (
   (for-all mf in f)
      (tModel.release>=mf.release))
selectCompatibles set of tModel from f: family of TModel given tm class of TModel
  in checkingSystem.tModels (
   (for-all tModel)
      (incompatiblewith(tModel,tm)=0)
selectCompatibleRoot tModel from selectedRoots: family of TModel
  in checkingSystem.tModels (
   (for-all root in selectedRoots)
      (tModel.tail >= root.tail))
```

4 Compatibility Checking by Ontology Categorization

In this section, we describe how the operator of `incompatiblewith` used in `selectCompatibles` in Listing 2 is realized by (1) using an ontology categorization technique to categorize important terms in the descriptions of Web services, and (2) assessing service compatibility according to semantic rules and the categorization results of terms. In Section 4.1, we first review the ontology categorization technique that has been designed in our earlier work [17].

4.1 Basics of Ontology Categorization

Web service information resources are documents that contain descriptive information about the semantics of Web services. We focus on semi-structured documents, including WSDL and OWL-S documents, as sources for checking their compatibility. We are interested in the portion of service descriptions that carries semantics related to the domain and will be useful in checking service compatibility. In WSDL documents, we use the names of *service interfaces*, *operations*, *input*, *output*, and *elements* of input and output, and about implementation descriptions in terms of *binding* protocols. In OWL-S documents, we have used *ServiceProfile* and *ServiceGrounding*, which we consider most relevant to domain knowledge.

Terms are first extracted from the service descriptions. Next, pre-processing is performed on the extracted terms. After that, the terms are used to establish a probabilistic model for ontology instance categorization. The categorization model is adapted from existing classification techniques found in the information retrieval area. It consists of two parts: (1) SVMV-based and (2) co-occurrence-based. Both parts of the model perform categorization based on the similarity ranking of terms in an independent manner. Their results are then merged using a simple voting scheme. The model is used to decode heterogeneous descriptions continuously and enhance the categorization basis by learning new service descriptions.

We have adapted SVMV, as a model of probabilistic text categorization, to derive the probability that a description d is categorized into a category c, i.e., $p(c|d)$. We adapt SVMV by introducing relationship patterns that pertain to service descriptions. For example, we consider the relationship between `InterfaceOperationType` and `input`, `InterfaceOperationType` and `output`, and `InterfaceType` and `Operation`.

In co-occurrence analysis, a description document is represented by a matrix of co-occurrence frequencies. We establish asymmetric links for each pair of terms. A cluster function defines the term similarity weights by the combined weight of both term t^j and t^k in document i and the inverse document frequency. To perform co-occurrence analysis, the asymmetric co-occurrence analysis function [17] is adapted for service descriptions. Our adaptation mainly concerns the definition of similarity weights and the calculation of the weighting factor.

4.2 Compatibility Checking

The *incompatiblewith* operator is evaluated by our system in the following way: categorize the values of a number of selected key tags in the corresponding XML constructs within the service descriptions and analyze the categorization results according to semantic rules. The key tags are selected from the conceptual definitions of services in a given upper service ontology, e.g., OWL-S. Table 1 lists all the objects and their key tags. Please notice that this forms a hierarchy of relevant tags for `Operation`, `TModel`, and `BusinessService`, which are consistent with the hierarchy of the checking system model described in Section 3.

For `Operation` objects, `operation names`, `input element names`, and `output element names` are the considered tags. For `TModel` objects, in addition to the name itself, all operations referred to by the definition of this `TModel` shall also be considered. For `BusinessService` objects, in addition to the name of this `BusinessService`, all `TModel` objects that are referred to by this `BusinessService` object are also included. We can see the tags are identified in a recursive way.

Table 1. Objects and their Key Tags

ObjectClass	*KeyTags*
Operation	(Operation) name
	Input element name(s) and types(s)
	Output element name(s) and types(s)
TModel	(TModel) name
	Operations
BusinessService	(BusinessService) name
	TModels

We use the following two sets of recursive rules to check compatibility of two objects of the same class using the ontology categorization tool summarized in Section 4.1. A rule engine based on the semantic service description language SWRL is used for implementing and executing the rules.

Rule set 1, which is used to deal with primary objects = {

Rule 1: For a particular tag that is simple and single valued, (primary) objects A and B that directly contain this tag are compatible bi-directionally regarding this tag if and only if their corresponding values of that tag are categorized together.

Rule 2: For a particular tag that is simple and multi-valued, (primary) object B is compatible to (primary) object A regarding this tag if and only if object B's value set is a subset of object A's value set, where both A and B directly contain this tag.

Rule 3: *incompatiblewith* on primary object B and primary object A is evaluated to be 0 if and only if regarding all selected tags object B is compatible with service A.}

Rule set 2, which is used to deal with complex objects = {

Rule 4: For a particular tag that is of simple type and single valued, (complex) objects A and B that directly contain this tag are compatible bi-directionally regarding this tag if and only if their corresponding values of that tag are categorized together.

Rule 5: For a particular tag that is of simple type and multi-valued, (complex) object B is compatible to complex object A regarding this tag if and only if object B's value set is a subset of object A's value set, where both A and B directly contain this tag.

Rule 6: *incompatiblewith* on complex object B and complex object A is evaluated to be 0 if and only if (1) for each selected containing simple tag, object B is compatible to object A regarding the tag, and (2) for each containing complex tags that are present in both objects, *incompatiblewith* on object B and object A is evaluated to be 0 and (3) for all selected containing complex tags, the constituent object set of object B is a superset of the constituent object set of object A.}

We refer to *primary objects* as objects defined in the checking system model that do not contain other objects defined in the model. *Complex objects* are objects defined in the checking system model that contain other objects defined in the model. Referring to Table 1, objects of `Operation` are primary objects and those of `TModel` and `BusinessService` are complex objects. In set 1, rule 1 is used to check the compatibility of each pair of simple single-valued attributes that belongs to the two primary objects to be compared. Rule 2 is used to check the compatibility of each pair of simple multivalued attributes that belong to two primary objects to be compared. Rule 3 is used to determine if one primary object is compatible with another primary object based on the result of rule 1 and rule 2.

In set 2, rule 4 is used to check the compatibility of each pair of simple single-valued attributes that belongs to the two complex objects to be compared. Rule 5 is used to check the compatibility of each pair of simple multi-valued attributes that belongs to the two complex objects to be compared. Rule 6 is used to determine if one complex object is compatible with another complex object based on the result of rule 4 and rule 5. For complex object a to be compatible with complex object b, all simple attributes of a must be compatible with their counterparts in b and all its complex attributes must be compatible with their counterpart in b or such complex attributes in a are not in existence in b. The relationship `compatible with` is unidirectional. To give an example, let us assume there are s1 and s2 both of `BusinessService`, and o1 and o2 both of Operation in the checking system. If o1 constitutes s1 and o1 and o2 constitute s2, s2 is compatible with s1. But s1 is not compatible with s2.

4.3 Incompatibility Factors

Our compatibility-checking system is designed to not only assess the compatibility of the objects, but also output the incompatibility factor(s) if needed. In Table 2, we summarize independent factors that may affect the compatibility of

Table 2. Independent Factors that May Affect the Compatibility of Web Services

ObjectClass	*ID*	*Factor*	*Compatibility*
Operation	CN	Change (Operation) Name	P
	CEN	Change Input and Output Element Name(s)	P
	CET	Change Input and Output Element Types(s)	NP
	RAE	Remove/Add Input and Output Element(s)	NP
TModel	CTN	Change of (TModel) Name	P
	AO	Add new Operation	P
	RO	Remove Operation	NP
	CO	Compatibility of Operation(s) referred by this object	
BusinessService	CBN	Change of (BusinessService) Name	P
	CE	Change of (BusinessService) Endpoint	NP
	CT	Compatibility of TModel(s) referred by this object	

Web services. These factors are referred to as *individual factor*. We have relaxed the definitions of compatibility and incompatibility from traditional software engineering, because services are no longer developed by a single development team as proprietary software assets. `P` in the table represents compatible and `NP` represents incompatible. If the compatibility cannot be determined, but is dependent on the compatibility of the factor itself, the cell is left blank.

We have designed the following rule set, i.e., Rule set 3, which is used to judge the factor that has caused incompatibility of two service objects, referred to as *incompatibility factor*.

Rule set 3 = {

Rule 7: For a particular tag that is of simple type and single valued, objects A and B that directly contain this tag are incompatible bi-directionally and this tag is output as a causal incompatibility factor, if and only if their corresponding values of that tag are not categorized together.

Rule 8: For a particular tag that is of simple type and multi-valued, object B is incompatible to object A and this tag is output as the causal incompatibility factor if and only if object B's value set is a superset of object A's value set, where both A and B directly contain this tag.

Rule 9: Complex object B is incompatible to complex object A if and only if for any containing complex tags that are present in both objects, the constituent object of object B is incompatible to object A and the causal incompatibility factor of the constituent objects will be output as one casual incompatibility factor for objects A and B.}

In set 3, rules 7 and 8 are used to identify incompatibility factors contributed by the simple attributes directly contained in the service objects. In particular, if the service objects are not categorized together on a tag, this tag is one casual incompatibility factor of the two objects. Rule 9 is used to identify incompatibility factors contributed by their containing service objects. In other words, incompatibility factors will propagate from one class of objects to another, as far as the latter has composition or aggregation relationships to the former. Let

us assume again there are s1 and s2 both of `BusinessService`, and o1 and o2 both of Operation in the checking system. o1 and o2 are constituent objects of s1 and s2, respectively. If f1 is a causal incompatibility factor between o1 and o2, it is also a causal incompatibility factor between s1 and s2.

5 Experiments and Analysis

We have considered in our experiments all the individual factors listed in Table 2 and the factors that are composed of possible combinations of multiple individual factors, referred to as *joint factors*. In our experiments, 600 Web services have been obtained from the following resources. We chose these resources because they are available publicly and are representative of different service domains. From these original Web services, we have formed a collection of 3200 Web services. These services are all mutations of the original services.
The Web service directories and indices are listed below:

- www.xmethods.com,
- www.bindingpoint.com,
- www.webservicelist.com,
- www.servicesweb.org/rubrique.en.php3?id_rubrique=14
- Web sites of four companies that publish their Web services: eBay, Amazon, PayPal, and http://www.xignite.com/

For all experiments, Web services are selected to form sets of various sizes. For each set, we have selected services in such a way that among all pairs within a set (1) for 20% of the pairs, the first service is compatible with the second service, and (2) for the remaining 80% of the pairs, the first service is incompatible with the second service. We imagine as more and more service providers contribute to the choices for services, which results in an increase of competition of the service market, the ratio of compatible services to incompatible services will also increase. Therefore, we not only test the ratio of 20%-80%, we also test 30%-70%, and 60%-40%. Distributions of both compatible and incompatible objects over all possible factors including individual and joint factors generally follow a uniform distribution. The detailed distributions of service compositions over incompatibility factors are listed in Table 3. Due to space constraints, we only show a part of our experiments and results.

We refer to the sets of service objects for testing the performance of compatibility checking in our experiments as a service set. In our experiments, we always create 10 different service sets so that the same experiment can be repeated 10 times. The result shown is the average performance over all service sets. As part of the preparation of the experiments, we partition a service set into a *base set* and *comparison set*, which comprise 30% and 70% of all the objects in the service set, respectively. A compatibility checking task is defined as the following: for a service picked from the *base set*, retrieve all services in the *comparison set* that are compatible with it and output the incompatibility factors of incompatible services.

Table 3. Distributions of Service Compositions over Incompatibility Factors

ObjectClass	*IndividualFactorID*	*JointFactorID*	*Per −cen −tile* (%)						
			A	B	C	D	E	F	G
Operation	$CN(notCEN)$		7	0	20	5	10	7	10
	$CEN(notCN)$		7	30	20	5	10	7	10
		$CN - CEN$	6	0	20	10	0	6	0
	CET		30	23	13	40	40	20	40
	RAE		30	24	13	40	40	20	40
		$CET - RAE$	20	23	14	0	0	40	0
$TModel$	CTN		5	0	15	3	7	5	5
	AO		5	20	15	4	8	5	5
		$CTN - AO$	5	0	15	8	0	5	5
	RO		40	30	20	60	60	20	60
		$CO - P$	5	10	20	5	5	5	5
		$CO - NP$	40	40	15	20	20	60	20
BusinesssService	CBN		10	15	30	5	15	10	10
	CE		40	35	20	60	60	20	60
		$CT - P$	10	15	30	15	5	10	10
		$CT - NP$	40	35	20	20	20	60	20

We perform experiments to measure the performance of the compatibility checking against different compositions of compatible and incompatible factors in service sets of 2000 service objects. We performed 600 (i.e., 30% of 2000) tasks per service set and there are all together 6000 tasks across all 10 service sets. In particular, we study how our checking system performs against different ratios of individual compatibility and joint compatibility factors. We draw *precision* and *recall* in percentile for compatible service retrieving, where Precision is represented as number of true compatible assessments/(number of true compatible assessments + number of false compatible assessments) and Recall is defined as the number of true compatible assessments+/(number of true compatible assessments + number of false incompatible assessments). The result is shown in figure 1. In the left part of figure 1, three lines give the performance over the compositions for column of Percentile A, B, and C in Table 3. As the compatibility ratio increases, the line moves right and up depicting an improved effectiveness of the system (precision of 13% for A compared with precision of 18% for C for recall of 100%). Although this result is straightforward, we use it to show that the system shall show more promising results with the further adaptation of SOA and component based application building. In particular, we observe that our technique favours data sources with a relatively large portion of compatible service objects and that are less skewed. This makes it suitable for service compatibility checking tasks, because service data objects are very likely to follow such a pattern.

In the right part of figure 1, three lines represent the performance over the compositions listed in A, D, and E of Percentile column. We are using this experiment to show that as the incompatibility and compatibility factors move

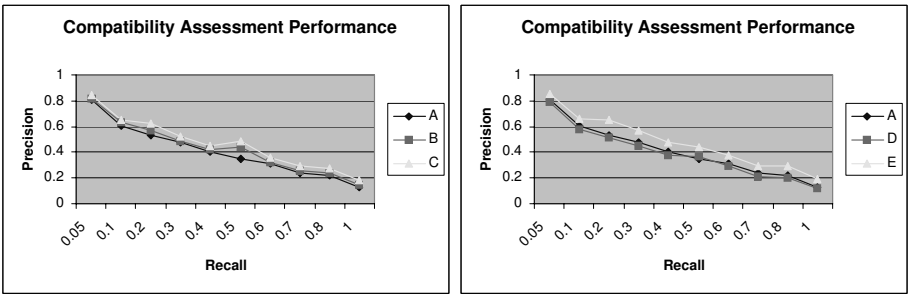

Fig. 1. Compatibility assessment performance for A, B, C, and A, D, E in table 3

from single to joint, or from simple to complex, the performance of the checking system degrades. For example, for recall rate of 100%, precision for E, A, and D are 19.6%, 13%, and 12.7%. As we can see, among E, A, and D, E has the highest percentage of individual compatibility factors and lowest (zero) percentage of joint compatibility factors. D has the highest percentage of joint compatibility factors. Joint compatibility factors on one hand shall increase the possibility that services are assessed as false negative due to the compatibility assessment mechanism of our design, and therefore, result in loss of the recall. On the other hand, it decreases the possibility that services are assessed as false positive and, therefore, results in improved precision. Service users are probably more concerned with precision. We can conclude that with higher percentage of individual factors, our technique tends to provide results that are more likely to satisfy service users in compatibility checking.

We also give the average correct identification rate for incompatibility factor identification in figure 2. The average correct identification rate is defined as (identification tasks percentage of identified factors)/number of identification tasks. The three bars in the left part of figure 2 represent the identification performance over the compositions listed in A, B, and C of Percentile column of in Table 3. As the data sources become less skewed, i.e., the ratio of compatible and incompatible objects increases, the categorization technique performs better; therefore, the correct identification rate improves as well (from 73.3% to 80.1%). In the right part of figure 2, the bars represent the identification performance over the compositions listed in F, A, and G of Percentile column of in Table 3. As the ratio of objects with joint incompatibility factors decreases, the complexity of identification of incompatibility is reduced because fewer tags are required to be categorized exactly. In this case, some identification results that used to earn only partial credits due to the wrong categorization of one joint factor can now earn full credits. Therefore, the system shows a better identification rate. In the figure, the rate increases from 70.2% to 82.6%. This result provides a useful implication to companies practicing service evolution: For the purpose of better service compatibility checking, incompatible changes are recommended to be introduced one-by-one. It is not recommended to wait a long while and then to introduce them in a batch process.

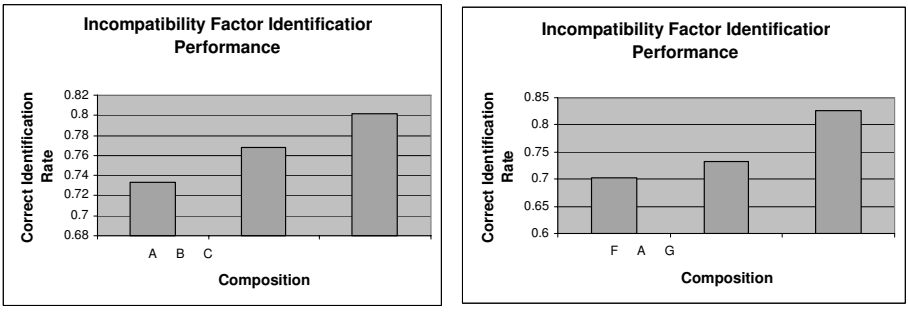

Fig. 2. Incompatibility factor identification performance for A, B, C and A, F, G in Table 3

6 Conclusions

In this paper, we describe an approach to checking Web service compatibility for Web service configuration management based on categorizing domain ontology instances extracted from service descriptions. The design details of a system model for Web service compatibility checking and the key operator for evaluating compatibility within the model are discussed. The contribution of the paper is to consider the engineering aspect of Web services that are due to autonomous and distributed design and development of the services, i.e., to meet the challenge within a service configuration system framework of engineering service compatibilities that may result from changes or parallel and independent service creation by other parties.

We are in the process of improving the basic categorization tool. Interesting issues include how to make it even more effective for different types of service descriptions, how to transfer such performance improvement to the compatibility checking system, and how service users' preferences can be incorporated into the categorization tool and the compatibility checking system.

References

1. Huhns, M.N.: A Research Agenda for Agent-Based Service-Oriented Architectures. In: Klusch, M., Rovatsos, M., Payne, T.R. (eds.) CIA 2006. LNCS, vol. 4149, pp. 8–22. Springer, Heidelberg (2006)
2. Roekind, M.J.: The source code control system. IEEE Trans. on Software Engineering 1(4), 364–370 (1975)
3. Kramer, J., Magee, J.: The Evolving Philosophers Problem: Dynamic Change Management. IEEE Trans. on Software Engineering 16(11) (1990)
4. Gutierrez, R., Mendoza, B., Huhns, M.N.: Behavioral Queries for Service Selection: An Agile Approach to SOC. In: Proc. IEEE International Conference on Web Services. IEEE Press, Salt Lake City (2007)

5. Schmerl, B.R., Marlin, C.D.: Versioning and consistency for dynamically composed configurations. In: Conradi, R. (ed.) ICSE-WS 1997 and SCM 1997. LNCS, vol. 1235, pp. 49–65. Springer, Heidelberg (1997)
6. Brown, K., Ellis, M.: Best practices for Web service versioning, http://www.ibm.com/developerworks/webservices/library/ws-version
7. Anand, S., et al.: Best Practices and Solutions for Managing Versioning of SOA Web Services, http://webservices.sys-con.comread/143883.htm
8. Frank, D., Lam, L., Fong, L., Fang, R., Vignola, C.: An Approach to Hosting Versioned Web Services. In: Proc. IEEE International Conference on Services Computing. IEEE Press, Los Alamitos (2007)
9. Do, H.H., Melnik, S., Rahm, E.: Comparison of schema matching evaluations. In: Proc. workshop on Web and Databases. IEEE Press, Los Alamitos (2002)
10. Madhavan, J., Bernstein, P.A., Rahm, E.: Generic Schema Matching with Cupid. In: Proc. of the 27th VLDB Conference. Springer, Heidelberg (2001)
11. Huang, J., Dang, J., Huhns, M.N.: Ontology Reconciliation for Service-Oriented Computing. In: Proc. IEEE International Conference on Services Computing. IEEE Press, Los Alamitos (2006)
12. Doan, A., Madhavan, J., Dhamankar, R., Domingos, P., Halevy, A.: Learning to match ontologies on the Semantic Web. The VLDB Journal 12, 303–319 (2003)
13. Nierstrasz, O., Gibbs, S., Tsichritzis, D.: Component-oriented software development. Communications of the ACM 127 (1992)
14. Yellin, D.M., Strom, R.E.: Protocol Specifications and Component Adaptors. ACM Trans. on Programming Languages and Systems 19(2), 292–333 (1997)
15. Georgiadis, I., Magee, J., Kramer, J.: Self Organising Software Architectures for Distributed Systems. In: Proc. the first workshop on Self-healing systems (2002)
16. Liu, Y.D., Smith, S.F.: A Formal Framework for Component Deployment. In: Proc. the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, pp. 325–344 (2006)
17. Liang, Q., Lam, H.: Web Service Matching By Ontology Instance Categorization. In: Proc. International Conference on Services Computing. IEEE Press, Los Alamitos (2008)
18. Wiebe, D.: Generic Software Configuration Management: Theory and Design. PhD thesis, published as Technical Report 90-07-03. Department of Computer Science, University of Washington (1990)