Research Journal of Applied Sciences, Engineering and Technology 9(11): 926-934, 2015
DOI:10.19026/rjaset.9.2585
ISSN: 2040-7459; e-ISSN: 2040-7467

Submitted: September 18, 2014 Accepted: October 12, 2014 Published: April 15, 2015

Research Article

Versioning Based Dynamic Reconfiguration for SOA Applications

Vallidevi Krishnamurthy
SSN College of Engineering, Anna University, Chennai-603110, Tamil Nadu, India

Abstract: Service Level Agreement or contract is a document that captures the functional and QoS levels agreed between the service provider and the consumer. In a service-oriented environment, individual services can be suitably composed to create a composite service. Whenever a new version of a composite service is created, for the same service consumer, in order to satisfy the change in consumer requirements, the respective contracts also need to be versioned. Likewise, whenever a service consumer is dynamically provided different versions of the services based on their requirements, the respective contracts also need to be activated, automatically. In this context, this study proposes an approach for, dynamic reconfiguration of a service-oriented application, which has been offered as a composite service with its corresponding version of the contract. This dynamic reconfiguration approach has been tested by applying it to a sample SOA based e-Shopping application.

Keywords: Backward-compatibility, change, contract, requirements, service-consumer, service-provider

INTRODUCTION

Service-Oriented Architecture (SOA) (Newcomer and Lomow, 2005; Erl, 2009) is an architectural style for developing software applications that use services as their building blocks. Web services (Schmelzer *et al*., 2002; Josuttis, 2007) are application components which are self-contained, self-describing and are used by other applications. These are both platform and language independent. The service providers could either provide an atomic service or a composite service. Whenever a service provider wants to provide a composite service, the provider has an option of composing the atomic services which are either developed in-house or outsourced. In the present work, it has been assumed, that the service provider composes the application, entirely with services that are developed in-house. Whenever, there is a change in the requirements posed by the service consumer, the provider creates a new version of the composite service. Based on these changing requirements, the provider should dynamically reconfigure the service oriented application by enabling the consumers, to switch among the relevant versions of the application. Whenever a different version of the application is switched to, it is essential that the corresponding contract is also simultaneously activated.

In this context, this study has made the following contributions:

- An approach for dynamic reconfiguration of the service oriented application by switching among its various versions along with the corresponding contracts.
- Automatic generation of the version numbers for the contracts. The audience for this study are the service oriented application providers who built applications with atomic services that were developed in-house.

Whenever additional features are requested by the service consumer, the service provider incorporates these features and creates a new version. Whenever a new version of the application is created, the service provider should check whether the newly developed version is a backward compatible one. The new version of the application is backward compatible, when the existing service consumers are not affected by the new version of the application. In that case, the application does not remove any of the existing features. Either additional features are included in the new version, or an alternative implementation for the existing service interface is provided. As this versioning of the composite service oriented application is not in the present scope of this study, only brief details are provided. More detailed explanation about the versioning of the service oriented application can be found in (Novakouski *et al*., 2012; Brown and Ellis, 2004; Evdemon, 2005; Kaminski *et al*., 2006; Hummer *et al*., 2011; Bianco *et al*., 2008).

Whenever the requirements of service consumers change, the service oriented application has to be dynamically reconfigured by switching among the various relevant versions along with the corresponding contracts.

Hence, contract versioning has to be realized, corresponding to different versions of the application.

In this study, an approach for automatically generating these contracts with a proper version number, using the WS-Agreement standard has been proposed.

A contract is the most important metadata in SOA (Schmelzer *et al*., 2002). Whenever a new version of the contract, does not affect the other existing service consumers, then the contract is a backward compatible one. In this study, the term, service provider is used for the one, who provides the composite service, to maintain a portal for a specific business organization. The term service consumer refers to the one who maintains a portal for a specific business organization. The term end user refers to the one who uses the specific business organization portal for some purpose.

LITERATURE REVIEW

Versioning of contracts can be achieved by three different ways (Erl *et al*., 2008).

Changing WSDL definition: Changing the WSDL document will have the most visible impact, than changing the message definitions and policies. When the first version of the WSDL definition is generated, version number is zero. A backward compatible change in the WSDL definition increments the minor version number and does not change its target namespace. An incompatible change in the WSDL definition, increments the major version number and stores that in a new target namespace. These concepts are explained through the following examples:

- **Adding a new operation:** When a WSDL definition is already implemented and in use, consumers will have dependencies on existing operation definitions. Extending the contract by adding a new operation in the WSDL definition will not impact these dependencies and is considered to be a backward compatible change.
- **Renaming an existing operation:** If the value of the name of an existing operation element has to be changed, then this is an incompatible change as the other existing service consumers will be affected because of this change. There are two common ways to handle this change.
  - **Force a new major version of contract:** Whenever the existing operation name has to be modified, then the contract is subject to an incompatible change that will require a new version of the contract.
  - **Add the renamed operation to the existing contract:** The new operation has to be added along with the original operation. This allows overlapping functionality to exist in the same service contract.

Changing message schema: Whenever an XML Schema definition undergoes a change that requires a new target namespace, then for that change in the schema, the change will propagate to the WSDL level, resulting in a new target namespace for the WSDL definition.
The common change types are:

- **Adding a new schema component:** A new element declaration will be added to the existing schema, which results in changing the namespace value.
- **Removing an existing schema component:** The removal of the component declaration from an existing XML Schema definition results in an incompatible change that forces a new schema and WSDL definition version.
- **Renaming an existing schema component:** By default, changing the value of an existing component, results in an incompatible change that requires a new target namespace and a new contract version.
- **Modifying the constraint of an existing schema component:** Adjusting the validation component is one possible type of changing constraints. Maximum occurs and minimum occurs can be specified as unbounded.

Changing policy assertions: Web service contract can be versioned with policies that express additional constraints, requirements and security attributes. All these attributes relate to the behavior of services. Human readable and machine readable policies can be created.

MATERIALS AND METHODS

Related work, application scenario and the proposed work are discussed under this section.

Related work: Dynamic Reconfiguration in SOA applications has been discussed in several existing works. However, many of these works focus on the dynamic reconfiguration with substitution of an equivalent atomic service.

Even in the few works which focus on the reconfiguration of the composite service, the corresponding change in contract versions is not addressed. However, in the proposed work, reconfiguration of the application along with the corresponding contract is the main focus, as the contract also varies, whenever the application is reconfigured.

In all the existing works, the service substitution is also realized with the services provided by different service providers. As the versioning based dynamic reconfiguration for a composite service, along with the corresponding contract, is the main focus of the proposed work, whenever the composite service switches, from one version to another, its corresponding contract also needs to be activated. Hence, if the composite service is composed of services, that are provided by multiple service providers, then, the contract corresponding to that composite service, will in

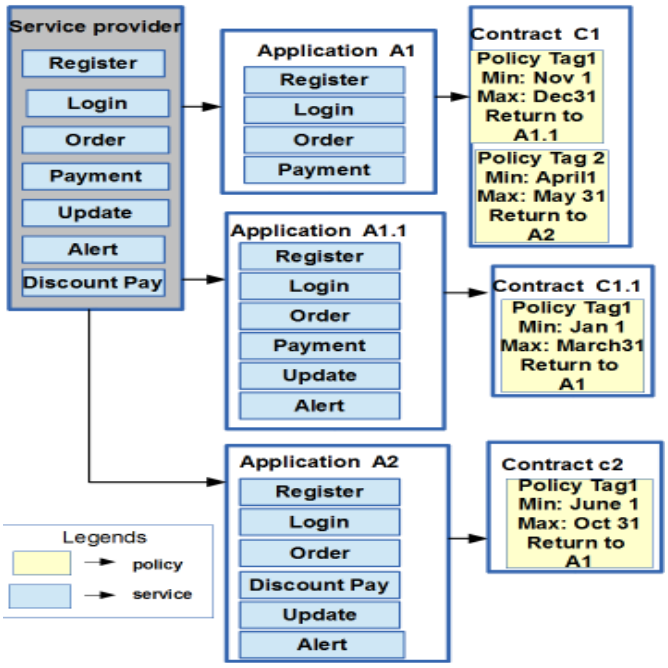

Fig. 1: e-Shopping application with various versions and their respective contracts

turn depend on various other service providers. Whenever, more dependency exists among various service providers, the complexity involved in the dynamic reconfiguration of the application increases. Hence, the present work, is based on the assumption, that all the services are provided by the same service provider.

Few works that focus on dynamic reconfiguration in the SOA environment are listed below:

- Dynamic Reconfigurable ESB Service Routing (DRESR) (Bai *et al*., 2007) approach, allows the abstract routing table to be changed at runtime in which the service provider for each node, service composition logic and service integration topology can be changed. Though DRESR is also for dynamically reconfiguring the composite service in the SOA environment, the service provider changes, when the services are dynamically reconfigured, which is the main difference between the work proposed in this study and the DRESR.
- SIROCO middleware (Fredj *et al*., 2008) platform, deals with the reconfiguration of service orchestrations, for the unavailable services, by replacing an atomic service with an alternative service which provides the same functionality. In SIROCO middleware, an atomic service is substituted at runtime for a service that is unavailable.
- The work proposed by Geebelen *et al*. (2008), focuses on dynamic reconfiguration in a composite service, by defining the templates statically. Template based dynamic reconfiguration discussed by Geebelen achieves reconfiguration by altering the workflows at runtime, that are defined statically. In this study, the workflow related details are kept as separate modules and hence they are reusable also in future. However, the services that are alternatively substituted need not be from the same service provider, which is the difference from the proposed work.

E-shopping application scenario: Figure 1 represents the overview of the various versions available in an Online Shopping application. The various functionalities available in the application are represented by the atomic services chosen by the service consumer. The same application with various versions is shown in this figure. Application A_1 has functionalities such as registration, login, selecting the items, inserting and deleting the items from the cart, payment and updation. According to this application, which is considered as a basic version, new users can register and then login with their credentials. After successful login, to the online shopping site, the user selects the required items and moves them to the shopping cart. Total cost of all these items stored inside the cart is calculated with the permissible discount and the receipt is generated. Subsequently, the user makes the payment. Once the payment is completed, the shipping dates are notified to the user. Updation of the items in the stock list takes place simultaneously with the notification of the shipping details to the user. In Application $A_{1.1}$, an *alert* service is included along with the atomic services in application A_1. As there

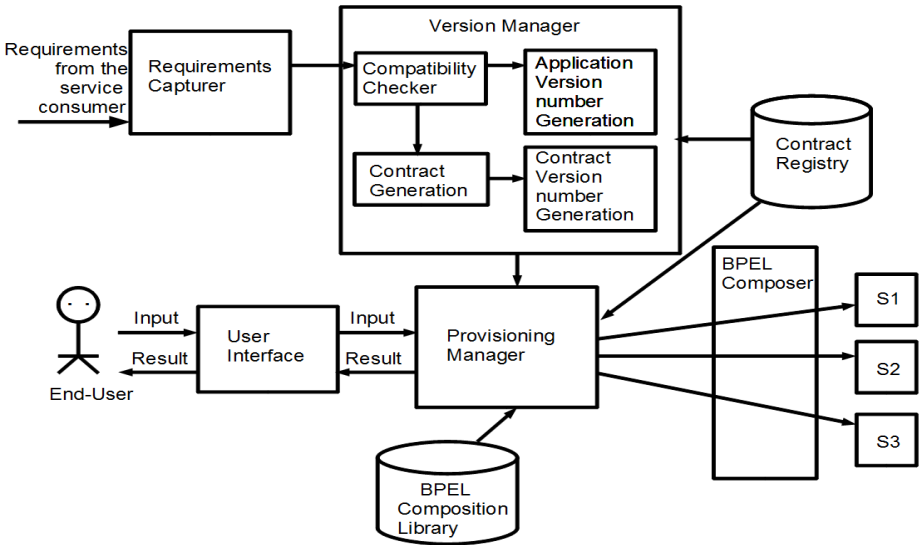

Fig. 2: System Architecture of V-DROPS

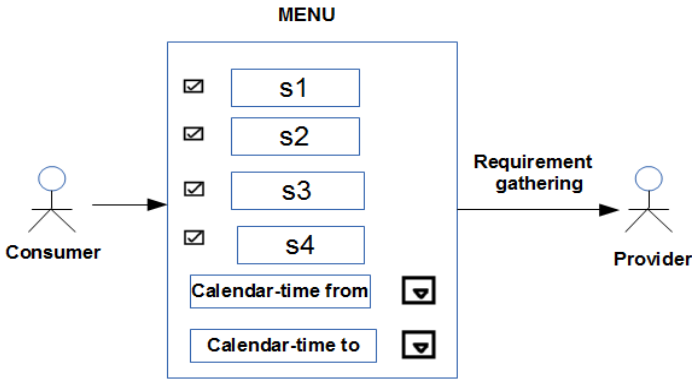

Fig. 3: Module Design for Requirements Capturer

is only an addition of service functionality (alert service) in application $A_{1.1}$, it is a minor version of application A_1.

In application A_2, which is shown in Fig. 1, Payment service is replaced by Discount Pay service. Also, there are two additional features (Security Check and Status Report) compared to the latest version of the application, $A_{1.1}$. As a service has been renamed, along with a change in its functionality and also two additional functionalities have been introduced, the application should have a major version change. This new version of the application is named as A_2. Security Check service collects the security information such as, pan card number from the customers when the payment exceeds rupees 50,000. Status Report is the service, which updates the delivery status information to the user, who has purchased the items. The service consumer switches among different versions of the application based upon the calendar period specified in the SLA. For example, whenever the application is expected to receive a huge number of requests, the service consumer might need the additional feature, provided by the alert service. This alert service provides an alarm message to the administrator when the total counts in the stock, drops down below a certain limit. This will help the administrator to refill the stock in the respective warehouse, which in turn helps to avoid scenarios leading to displaying the message "*Out of Stock*" for a particular product. In this case, the service consumer switches from application A_1 to application $A_{1.1}$. Otherwise, the service consumers can also request for application A_2 instead of $A_{1.1}$, as A_2 has additional features. All these requests are specified in the initial contract that is established between the service provider and the service consumer. In Fig. 1, the policy tags are also shown as examples, which specify the calendar period through which the various versions of the application are available. These policy tags are specified as a part of the service contract that is mutually agreed between the service provider and the service consumer. These calendar periods are obtained as input from the service consumers, which are explained in more detail through Fig. 2 and 3.

V-DROPS: Versioning based Dynamic Reconfiguration for a cOmPosite Service and its contract in an SOA environment: The system architecture is shown in Fig. 2. A brief description of this figure is as follows. The service provider develops various individual services such as (S1, S2, S3, S4) and these atomic services are provided through a menu to the service consumer. The service consumer can choose the required services and that request is collected by the Requirements Capturer. The selected services are composed by the service provider by the BPEL composition engine. The requirements captured by Requirements Capturer are then fed to the Version Manager. The Version Manager analyzes the version details from the log information and identifies the appropriate version number of the application. The Provision Manager maps the particular application with the respective contract and the end user can use the particular application in the specified calendar period. A service level agreement which is also called as the contract is generated by the service provider. Once it is mutually accepted, between the service consumer and the provider, the service consumer starts using the application.

There are four modules in the proposed V-DROPS approach. They are:

Requirements capturer: Menu driven lists of services developed by the service provider are offered to the service consumers. The consumer has to choose the various atomic services based on his requirements. The consumer also selects the duration for which these atomic services are required. Figure 3 shows the module design for Requirements Capturer. In this figure, the various atomic services namely S1, S2, S3 and S4 are selected. Followed by that, to specify the calendar period, two combo boxes "*calendar time from*" and "*calendar time to*" are used to obtain the calendar period as input, from the service consumer. All these collected requirements are fed to the *BPEL Composer*. For the entire duration of 12 months, the service consumer selects the required services. Whenever the selection of atomic service is missing for some duration, then, the composite service becomes inactive during that period.

Version manager: The requirements obtained through the Requirements Capturer are forwarded to the Backward Compatibility Checker, which is available inside the Version Manager. The Backward Compatibility Checker checks the log information that is available in the Contract Registry, to identify whether there is any composite service already being provided for this same service consumer. The latest contract details are retrieved from the Contract Registry and are fed as input to the parser. If there is no entry in the contract registry, the information related to the selected

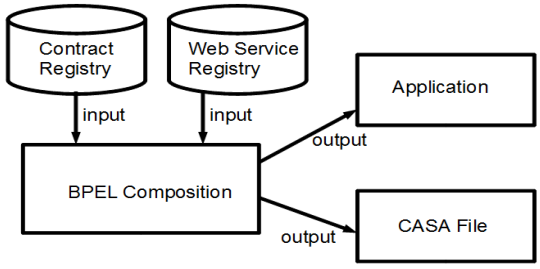

Fig. 4: Module Design for BPEL Composer

services are logged in the Contract Registry and the application is versioned as application A_1. This means that, it is the first time the composite service is being provided for the service consumer and there was no contract that was existing earlier between the same the service consumer and the provider. Otherwise, the selected atomic services are compared with the already existing application for the backward compatibility. If the selected sets of services are backward compatible with the atomic services of the already existing application, a minor version of the application is created. Whenever the backward compatibility is violated, then a major version of existing application is generated. Whenever the application is versioned, its respective contract is also generated and versioned. The new contract details are also logged in the Contract Registry.

BPEL composer: The requests from the service consumer are obtained and the services are composed through the BPEL composition. Figure 4 shows the design for the BPEL composition module. From contract registry, the atomic services selected by the service consumers are identified and those services are selected from the web service registry to compose them through BPEL. After composing the atomic services, the composite service and the Composite Application Service Assembly (CASA) file are obtained as output.

Provisioning manager: The Provisioning Manager maps the various versions of the application along with their respective contracts, which are available in the BPEL Composition Registry and Contract Registry. The version number of the application should match with the version number of the contract. Based on the request from the service consumer, the application and its respective contract are dynamically mapped. In Fig. 2, the Provisioning Manager is shown where the entries from the BPEL composition library and the contract library are mapped. Provisioning Manager compares the system date with the date specified in the contract and switches from the existing version to the next version whenever required. Thus, the Provisioning Manager enables dynamic reconfiguration of the application based on the requirements of the service consumer. The various inputs from the end-user are

forwarded to the respective application, based on the decision taken by the Provisioning Manager.

RESULTS AND DISCUSSION OF V-DROPS

This section explains the dynamic contract generation for the SOA application using the WS-Agreement standard. The main contributions of the study namely, dynamic reconfiguration of the application along with the corresponding contracts and automatic generations of contract version numbers are also explained in detail. These dynamic reconfigurations are based on the requirements of the service consumers specified in their SLA (Fig. 5).

Automatic contract generation: Contract generation can be realized using two different specification standards namely WSLA (Keller and Ludwig, 2003) and WS- Agreement (Andrieux *et al*., 2005). In the proposed work, WS-Agreement specification standard is used for the contract generation. Contract is dynamically generated using the JAXB marshalling process. The input values for the application are provided, by creating an appropriate BPEL workflow. On deployment of composite application, Netbeans IDE automatically generates a Composite Application Service Assembly (CASA) file. The whole configuration details and the details of the atomic services involved for the composite application are available in the CASA file. From the CASA file, the wsdl definition details are parsed using the DOM parser (Coyle, 2002). Later, the details corresponding to each of the atomic services are stored in separate string array and are included in the contract. The atomic services based on the requirements of the service consumer are selected and then the calendar period through which those services should be activated is specified as shown in Fig. 5. After submitting the calendar period, the version number and its respective contracts are generated automatically. The generated contract is shown in the left side of Fig. 5. The details specified by the service consumer are captured as input and they are also reflected in the contract. These details are marked in the red color.

Automatic contract versioning and dynamic reconfiguration: Versioning of contracts is performed based on the concept of backward compatibility. The most recent contract details are tracked from the contract registry and they are given as input to the DOM parser. Based on the backward compatibility between the existing contract and the currently generated one, the version number is generated for the new contract. For example, if there is a new contract C_n and an existing contract C_e, where *n* and *e* are version numbers, the value n is identified based on the backward compatibility checking between the existing and the new contract. Hence, the details of the atomic services in contract C_n that are stored in the string array (which was explained in the Automatic Contract Generation section), are compared with the string array corresponding to the contract C_e. If all the elements in the string array of contract C_e, are available in the string array of Contract C_n, then it is backward

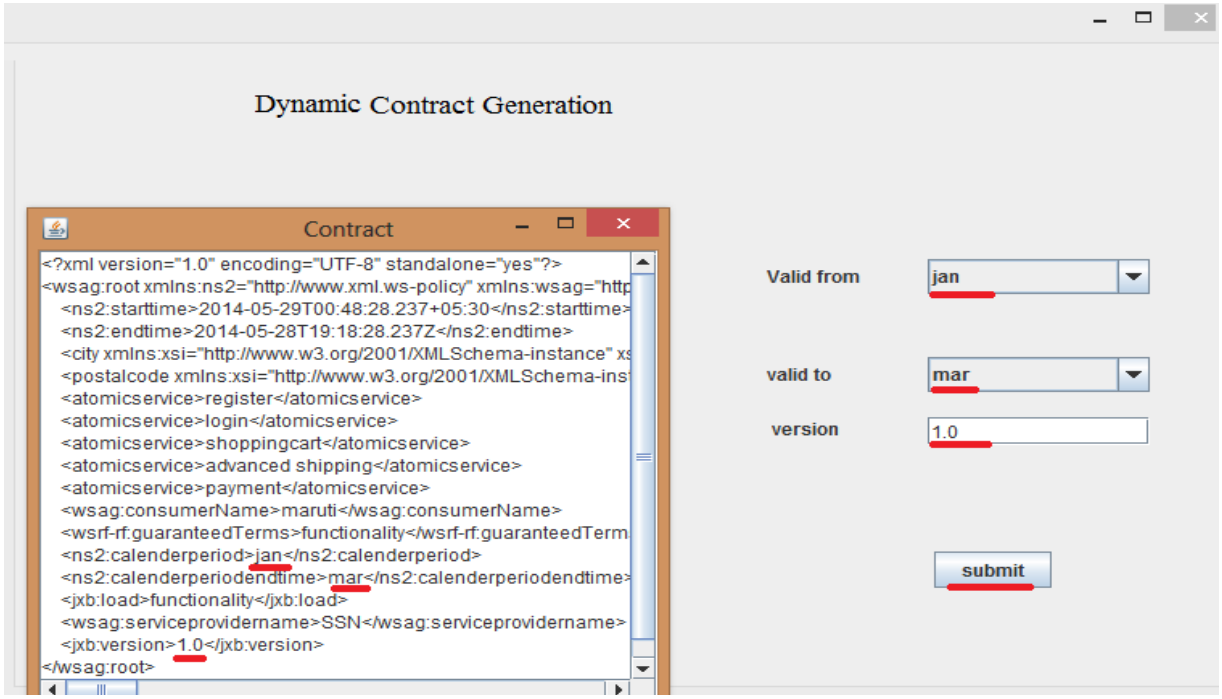

Fig. 5: Contract generation based on user selection

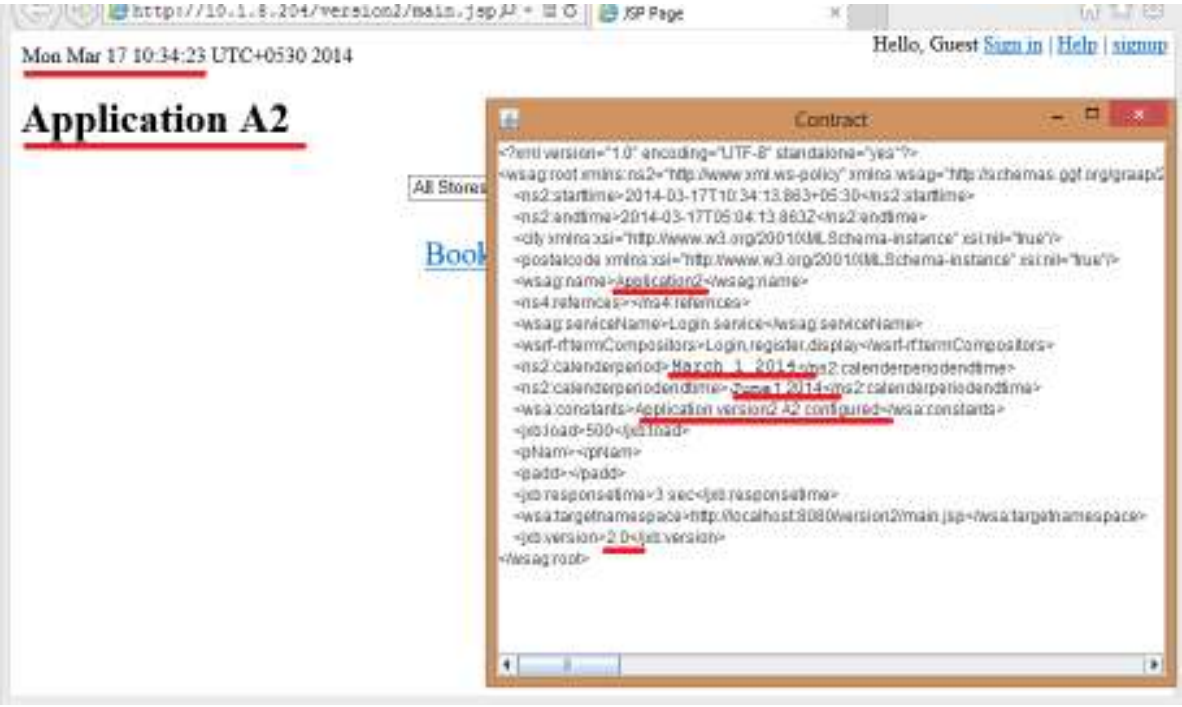

Fig. 6: Reconfiguration of application and respective contract

Table 1: Analysis report for backward compatibility checking

Changes made in	Supports backward compatibility	Violates backward compatibility
Wsdl definition	• Adding wsdl definition • Adding a new wsdl port type definition • Adding a new wsdl binding	• Renaming an existing wsdl definition. • Removing an existing wsdl definition.
Examples:	A1.1->Alert service is added	A2->payment service is altered as discount pay service.
Message schema	• Adding xml schema or attribute declaration • Reducing the constraint granularity of an xml schema element	• Renaming an optional or required schema element • Increasing the constraint granularity • Removing an optional/required schema element
Examples:	A1.1.1->An attribute security number is included. A1.1.2->Constraint granularity of attributes is changed as min = 0 max = unbounded. The default granularity for attributes will be min = 1 max = unbounded.	A1.2->Renaming the attribute name password as secured number /Removing the attribute name password.
Adding policy	• Adding a new ignorable policy assertion. • Adding a new policy alternative.	• Adding a required policy assertion.
Examples:	Min and max validity of particular service can be stated using policy assertion. Ignorable policies are always backward compatible. Alert service will be available only between 6-10 pm.	Required policy assertion violation the backward compatibility.

compatible. Hence, the contract C_n will be the minor version of the contract C_e.

In another case, where few elements in the string array of the contract C_n are missing or altered, then it violates the rule of backward compatibility. Hence, a major version of the contract is generated and the new application that corresponds to this contract also has a functionality that is removed/altered when compared to the existing version. By Changing the WSDL definition which was explained in Background section, major/minor version of the contract is generated. By changing the Message schema and the Policy Assertions minor/very-minor version of the contract is generated. The running system automatically switches to the application version, requested by the service consumer. This is explained in the screen-shot that is captured in Fig. 6, where the service consumer has requested for the Application A_2 which is shown in the contract with a red line. In this figure, application A_2 is requested for the duration from March 1, 2014 to June 1, 2014. The left side of Fig. 6, also contains the current time of the system in which the application is executed. March 17, 2014 lies between the calendar period specified in the contract and the requested application A2 is activated automatically.

The analysis report for the various changes in the application which leads to major/minor/very-minor versions of both the application and the contract are

captured in Table 1. This analysis report is based on the backward compatibility concept that was discussed earlier.

In the cases where the application supports the backward compatibility, it leads to minor/very-minor version of both the application and its respective contract. In some other cases where the newly created application does not support backward compatibility with the existing one, then a major/minor version of application and the contract are generated. The examples for the change in the WSDL definition, Message Schema and the Policy Assertions are provided in Table 1.

CONCLUSION AND RECOMMENDATIONS

Contract creates trustworthiness between various business partners. An automated contract generation and versioning of contracts for dynamic reconfiguration of the SOA application is discussed in this study. Versioning of the application for the same service consumer is needed as the consumer requests are changing frequently. Whenever the application is versioned, the respective contract also needs to be generated and versioned. The web service composition is based on the BPEL Engine and the contract generation is according to WS-Agreement specification standard. Both, versioning of the application and the contract are based on the concept of the backward compatibility. In the proposed work, atomic services developed by the same service provider, are considered for the composition of the SOA application. This is because, to dynamically reconfigure the SOA application, through the V-DROPS approach, services developed by the same provider, reduces the dependency on other service providers.

Dynamic composition of the services provided by various service providers, with the corresponding contract being activated at that time, will be the future direction. With this approach, the service consumers need not freeze all the requirements at the design time itself. They can request their requirements even during runtime.

ACKNOWLEDGMENT

I would like to thank, Dr. Chitra Babu, my supervisor, for her continuous encouragement, support and tireless guidance in shaping this study.

REFERENCES

Andrieux, A., K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke and M. Xu, 2005. Web services agreement specification (WS-agreement). Technical Report, Global Grid Forum, Grid Resource Allocation Agreement Protocol (GRAAP) WG.

Bai, X., J. Xie, B. Chen and S. Xiao, 2007. DRESR: Dynamic routing in enterprise service bus. Proceedings of the IEEE International Conference on e-Business Engineering (ICEBE'07). Hong Kong, China, pp: 528-531.

Bianco, P., G.A. Lewis and P.F. Merson, 2008. Service level agreements in service-oriented architecture environments, software architecture technology initiative integration of software-intensive systems initiative. Technical Note-CMU/SEI-2008-TN-021, Carnegie Mellon University.

Brown, K. and M. Ellis, 2004. Best Practices for Web Services Versioning. Retrieved from: https://www.ibm.com/developerworks/webservices/library/ws-version/.

Coyle, F.P., 2002. XML Web Services and the Data Revolution. 1st Edn., Pearson Education, India.

Erl, T., 2009. SOA Design Patterns. Prentice Hall PTR, USA.

Erl, T., A. Kamarkar, P.H. Haas, U. Yacinalp, C.K. Liu, D. Orchard, A. Tost and J. Pasley, 2008. Web Service Contract Design and Versioning for SOA. Prentice Hall, USA.

Evdemon, J., 2005. Principles of Service Design: Service Versioning. Retrieved from: “http:// msdn. microsoft. com/en-us/library/ms954726.aspx".

Fredj, M., N. Georgantas, V. Issarny and A. Zarras, 2008. Dynamic service substitution in service-oriented architectures. Proceedings of the IEEE Congress on Services-Part I (SERVICES'08). Honolulu, HI, pp: 101-104.

Geebelen, K., S. Michiels and W. Joosen, 2008. Dynamic reconfiguration using template based web service composition. Proceedings of the 3rd Workshop on Middleware for Service Oriented Computing (MW4SOC'08). Belgium, New York, USA, pp: 49-54.

Hummer, W., P. Leitner, A. Michlmayr, F. Rosenberg and S. Dustdar, 2011. VRESCO-Vienna Runtime Environment for Service-Oriented Computing. In: Service Engineering: European Research Results. Springer, Vienna, pp: 299-324.

Josuttis, N.M., 2007. SOA in Practice. Oreilly Shroff Publications, USA.

Kaminski, P., H. Muller and M. Litoiu, 2006. A design for adaptive web service evolution. Proceedings of the 28th International Workshop on Self-Adaptation and Self-Managing Systems (SEAMS'06). Shanghai, China, pp: 86-92.

Keller, K. and H. Ludwig, 2003. The WSLA framework: Specifying and monitoring service level agreements for web services. J. Netw. Syst. Manage., 11(1): 57-81.

Newcomer, E. and G. Lomow, 2005. Understanding SOA with Web Services. 1st Edn., Pearson, India.

Novakouski, M., G. Lewis, W. Anderson and J. Davenport, 2012. Best practices for artifact versioning in service-oriented systems. Technical Note-CMU/SEI-2011-TN-009.

Schmelzer, R., T. Vandersypen, J. Bloomberg, M. Siddlingaiah, S. Huunting, M.D. Qualis, D. Houlding, C. Darby and D. Kennedy, 2002. XML and WebServices Unleashed. 1st Edn., Sams Publishing, Pearson, India.