Institute of Architecture of Application Systems
University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3613

Extending a Multi-tenant Aware ESB Solution with Evolution Management

Penghao Tian

Course of Study:	Computer Science
Examiner:	Prof. Dr. Frank Leymann
Supervisor:	Dr. Vasilios Andrikopoulos

Commenced:	January 14, 2014
Completed:	August 15, 2014
CR-Classification:	D.2.7, D.2.12, H.4.0

Abstract

Services have been improved over time to meet the increasing demand of service consumers. A service could be changed at any state in service life cycle. An inefficient service version management with arbitrary decisions could lead to disconnections between service consumers and service providers. A service version control management system is therefore necessary.

The goal of this thesis is to specify, design and implement a service evolution management system for a multi-tenant aware ESB solution, so that the ESB could have the capabilities to manage the changes of multiple service versions between service consumers and service providers in a transparent manner. Furthermore, a function should be provided to check the compatibility between service versions, and, moreover, the calculation of service identifiers should be executed automatically, because service descriptions do not contain version-related information.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

1 Introduction

This diploma thesis is aimed at extending a multi-tenant aware Enterprise Service Bus (ESB) solution with service evolution management. In the following, the motivation and problem definition for this research are described and discussed. Outline of this work is given at the end of this chapter.

1.1 Motivation

Cloud computing is a new paradigm, which offers various computing resources to customers on demand. However, each service provider has to face a problem, "How should an application be designed to enable maximal sharing of resources among customers, so that the cost for each individual customer could be reduced?" Therefore, the multi-tenancy solution has been proposed in recent years to solve this problem. Making multi-tenant aware applications enables the computing capabilities to be better utilized by reducing underutilization and by sharing common code base and data, moreover, by automated sales and providing process.

Dominik Muhler has developed in his thesis [Muh12] a multi-tenant management and administration system based on the open source ESB solution Apache ServiceMix 4.3.0 for "Java Business Integration (JBI) multi-tenant multi-container Support". It uses a role-based control mechanism and two shared registries respectively for tenants and services to guarantee the data isolation between tenants. Moreover, this system has been designed as one part of a Software as a Service (SaaS) platform to ensure the elasticity characteristic of cloud computing [Muh12].

Services have to be improved over time because of increasing challenges in the business environment. A service could be changed at any state in service life cycle, by adding new functionalities, or modifying its data types, messages and operations. Several possible evolution paths of a service could be considered as follows. We assume that each service version is represented in format Major#.Minor#.

- A service provider registers a service version 1.0 and a service consumer employs this service version 1.0.
- The service provider might deploy a new compatible service version 1.1 to replace service version 1.0, the service consumer could still use the service version 1.0 or update it to 1.1. If the service consumer does use the previous version 1.0, its request messages have to be adapted for service version 1.1.

- The service provider might offer a new service version 2.0 in parallel with version 1.1. If the service version 2.0 is incompatible with 1.1, all consumers' requests of the versions 1.0 and 1.1 have to be targeted on service version 1.1, while the new version 2.0 is provided to new service consumers. If the version 1.1 is decommissioned by the service provider, all service consumers which use the versions 1.0 and 1.1 are required to update their services to the service version 2.0.

The service evolution could lead to the decommissioning of an existing service version or the appearance of multiple service versions of a service, so that the interaction between service consumer and service provider is possibly broken and unexpected effects can occur.

Although the system proposed by Dominik Muhler has provided simple functionalities to execute service registration and unregistration, it is not capable enough to deal with service evolution. Therefore, its capacities should be extended for this purpose.

1.2 Problem definition

In the first place, there is a need to investigate an approach to managing service evolution. Currently, a guide-line based approach is wildly used. But the subjective deduction from service developers and the lack of a solid theoretical foundation could result in errors. For this purpose, [ABP12] has defined a theoretical framework to guarantee a correct service versioning transition and service compatibility so that service changes can evolve consistently and transparently.

Based on the possible service evolution paths and the mechanism of management of service evolution, this thesis aims to develop a service version control management system for extending a multi-tenant aware ESB solution namely JBI multi-tenant multi-container Support (JBIMulti2) implemented by Dominik Muhler. The focus is on extending the version control capabilities of the ESB and the tenant-based version management is out of scope for this work.

To achieve this goal, the works about the JBIMulti2 system and service evolution must be deeply investigated. The requirements for the service version control management system could be thereby acquired. Furthermore, to explain how the JBIMulti2 system can be extended for service evolution management purposes, use cases based on service evolution paths should be conceived. A Version Registry should be introduced as a database for storing all version-related information. Finally, a dynamic routing mechanism is necessary to ensure correct routing between compatible service versions, since the service could be replaced by a compatible (or incompatible) version.

1.3 Outline

This thesis is divided into six chapters. They represent different research aspects which are listed as follows. Figure 1.1 shows the flow diagram of these chapters.

Chapter 2 – Background and Related Work: This chapter provides an overview of various conceptual and technological fundamentals such as Cloud Computing, Service-Oriented Architecture, Enterprise Service Bus, etc. and introduces the essential works which are necessary for this thesis.

Chapter 3 – Specification and Design: After the requirements of the service version control management are discussed at the beginning of this chapter, the functionalities of the system will be explained with three use cases. Some more topics, such as design of the system, formal expression mechanism of service descriptions and design for dynamic routing are also introduced in this chapter.

Chapter 4 – Implementation: The related technologies used to implement the extension of the JBIMulti2 system are introduced. The Version Registry, the Compatibility Checking Function (CCF) assessment function and the extended functionalities for managing service evolution are implemented in this chapter.

Chapter 5 – Validation: Firstly, the environment for validation is built. Secondly, the CCF assessment function is validated using a simple Web service use case which forms a simple service evolution path. Finally, the functionalities provided by JBIMulti2 are validated by means of the testing suite of soapUI.

Chapter 6 – Conclusions: This chapter summarizes the overall work and suggests possible extensions to the system in the future.

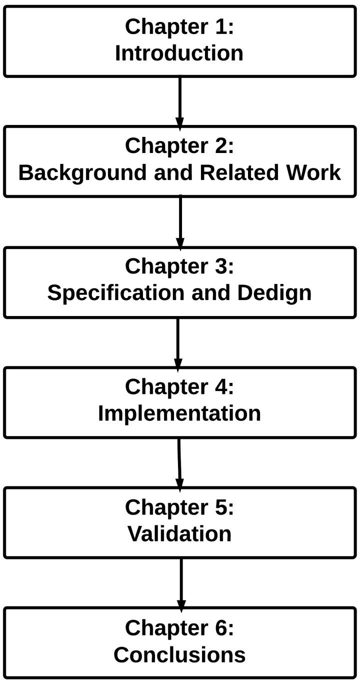


Figure 1.1: Flow diagram of chapters

1.4 Abbreviation

The following list contains abbreviations used throughout this document.

ESB	Enterprise Service Bus
JBIMulti2	JBI multi-tenant multi-container Support
CCF	Compatibility Checking Function
SOA	Service-Oriented Architecture
NIST	the National Institute of Standards and Technology
IaaS	Infrastructure as a Service
PaaS	Platform as a Service
SaaS	Software as a Service
IT	Information Technology
WSDL	Web Service Description Language
HTTP	HyperText Transfer Protocol
XML	Extensible Markup Language
JMS	Java Message Service
SMTP	Simple Mail Transport Protocol
TCP	Transmission Control Protocol
JBI	Java Business Integration
JCP	Java Community Process
JSR	Java Specification Request
NMR	Normalized message Router
SE	Service Engine
BC	Binding Component
CLOB	Character Large Object
NM	Normalized Message
OSGi	Open Service Gateway initiative
JVM	Java Virtual Machine
JAR	Java Archive
ESB^{MT}	Multi-tenant aware Enterprise Service Bus

UI	User Interface
API	Application Program/Programming Interface
UUID	Universally Unique Identifier
SCM	Software Configuration Management
VIDs	Version Identifiers
URL	Uniform Resource Locator
UUDI	Universal Description Discovery and Integration standard
ASD	Abstract Service Description
URI	Universal Resources Identifier
Java EE	Java Platform, Enterprise Edition
JAX-WS	Java API for XML-based Web Services
JAXB	Java Architecture for XML Binding
JSF	JavaService Faces
JDK	Java Development Kit
POM	Project Object Model
DDL	Data Definition Language
JPA	Java Persistence API

2 Background and Related Work

This diploma thesis is based on various conceptual and technological fundamentals. In this chapter we firstly provide an overview of these concepts and technologies for a better understanding of the basis of our work. After that, we introduce some works which are important foundations of this thesis, such as JBIMulti2, service evolution and the extension of the multi-tenant ESB solution for service version management.

2.1 Cloud Computing

Due to the rapid development, cloud computing has become a hot issue in Information Technology (IT) industry for many years. Various terms regarding cloud are everywhere, e.g. Cloud Storage, Cloud Bridge, Cloud Enabler, etc., which make people confused about the real meaning behind it. For better understanding of cloud computing, the National Institute of Standards and Technology, namely NIST, has proposed the definition of cloud computing: "cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g. networks, servers, storage, applications and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" [MG11]. This definition has been widely recognized.

NIST proposed five key characteristics [MG11].

- On-demand self-service
  Cloud users can easily access computing resources as needed. The service provider does not need to interfere with the user.
- Broad network access
  Cloud users can access services over the network through standard mechanisms.
- Resource pooling
  Computing resources of a provider are pooled and use a multi-tenant model to serve multiple consumers, so that the consumer generally has no sense of control and knowledge over the exact location of provided resources but may specify location at a higher level of abstraction.
- Rapid elasticity
  The available resources can be rapidly increased and decreased as needed, so that the consumer can provision them in any quantity at any time.

- Measured Service
  According to specific demands for services, cloud system measures and prices the usage of resources. Resource usage is transparent for both provider and consumer.

Among the five essential characteristics, the broad network access is the hardware foundation, on-demand self-service is the goal and the others are the means for achieving the goal.

According to the definition of NIST, the cloud computing service models consist of SaaS, PaaS and IaaS [MG11].

- Software as a service, or SaaS: where the consumer is able to employ applications that are provided by provider and running on a cloud infrastructure. The consumer does not need to manage the underlying infrastructure and platform where the applications run. Because these applications are installed and running on the provider side, the maintenance of the consumer side becomes easier.
- Platform as a Service, or PaaS: where the consumer is able to deploy own applications applying programming language, libraries, services and tools provided by provider. The consumer does not need to know the underlying infrastructure, where his own applications run, in order to control the applications.
- Infrastructure as a Service, or IaaS: where the consumer is able to use fundamental computing resources (e.g. network, storage) in addition to deploying own applications. The consumer controls only the used computing resources but not the underlying cloud infrastructure.

Finally, the NIST defines four deployment models [MG11]. The first model is private cloud. A single organization comprises multiple consumers owns or provisions the cloud infrastructure that is managed and operated by this organization. The second model is community cloud. It provides the organizations where the consumers have shared concerns own or provisioned the cloud infrastructure. The third one is public cloud. It works like an organization that provides its cloud computing capabilities to public owns or provisions the cloud infrastructure. The last model is hybrid cloud. This model combines two or more distinct deployment models as mentioned above.

The cloud is leading the future trend. Its flexibility and scalability of cloud computing resources and processing capabilities offer not only significant cost benefits, but also the enhanced ability for connecting customers, partners and providers like never before. However, without Service-Oriented Architecture, it is almost impossible to reach the cloud [RB14].

2.2 Service-Oriented Architecture

With the rise of the Internet, more and more enterprises attempt to transfer their businesses into independent, high scalable Internet-based services. The concept "Web Services" was proposed. The Service Oriented Architecture (SOA) and Web Services concepts are mutually influential: Web Services momentum will bring SOA to mainstream users and the best-practice architecture of SOA will help make Web Services successful.

SOA is a popular architectural paradigm for applications. It advocates that the functionality of a software component can be provided as a service and reused by other software components or services. The existing systems, applications and users thus can be integrated into a flexible architecture and the existing IT investments can be reused to adapt to changing needs and challenges. Cost reduction, delivering IT solution faster and smarter, and maximizing return on investments, these three drivers urge the widely utilization of the SOA approach nowadays.

There are different definitions about SOA, such as:

- The Service-Oriented Architecture was defined by the Open Group as "an architectural style that supports service orientation. Service orientation is a way of thinking in terms of services and service-based development and the outcomes of services" [Gro].
- Another definition is given by Service-Architecture.com: "a service-oriented architecture is essentially a collection of services. These services communicate with each other. The communication can involve either simple data passing or it cloud involve two or more services coordinating some activity" [Arc].

Obviously, service is the key concept of SOA. A Service is a well-defined, self-contained and reusable function. It does not depend on the context or state of other services. A service is available to other applications or Web Services through standard network and application interfaces and protocols [PTDL07]. With loose coupling, interoperability, efficiency and standardization, SOA provide flexibility and agility to organizations, so that these organizations are able to response to the rapid and unpredictable changes.

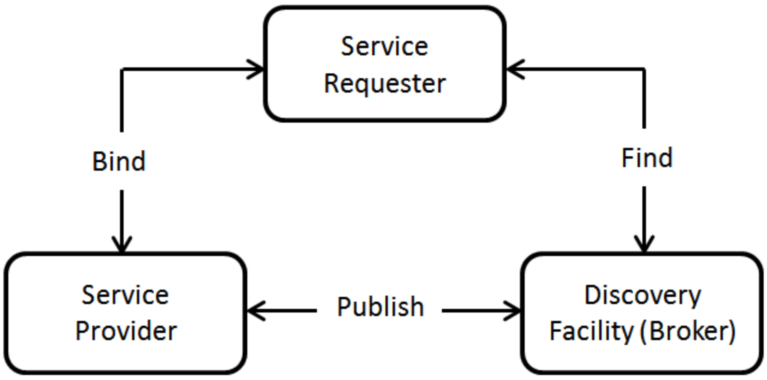


Figure 2.1: Publish, Find, Bind pattern

The Publish, Find, Bind pattern of SOA defines the operations among the requester, provider and broker. First, a description of a service has to be registered to a discovery facility (broker) by the service provider. A service requester accesses the discovery facility to inquire the information about the service what he wants and get the response with a concrete service endpoint. The requester can thus use the endpoint to bind his expected service and execute a business activity [WCL^{+}05].

A large-scale system is almost a heterogeneous system, in which each application has its own business processes using various technologies and protocols. These enterprise applications need a practical middleware to simplify and unify their interaction with each other. SOA enables reusing applications and services, and connecting resources. An Enterprise Service Bus reduces the complexity for integration of applications and services, and forms a foundation of SOA [PTDL07].

2.3 Enterprise Service Bus

Initially, enterprises have integrated their applications manually by point-to-point topology. This point-to-point approach delivers less flexibility and scalability, and becomes brittle and hard to manage over time. Thus, the Enterprise Application Integration (EAI) approach has emerged. Instead of directly passing messages to other applications, applications send messages to EAI. The EAI then has to forward the messages correctly to other applications. The centralized approach might have the problem of a single point failure. To avoid this weakness and realize the SOA, the Enterprise Service Bus approach has been introduced [Ort07].

Similar to EAI, an ESB is also considered as a mediation service for connecting enterprise applications and services. However, there are still two main differences between them. First, ESB products change the hub-and-spoke model in EAI products to a bus-based model. As a centralized architecture, a hub-and-spoke model employs a hub or broker to exchange data. The bus model uses a distributed architecture to implement its functionalities. Second, while EAI products are mainly based on proprietary technologies, ESB products are based on open standards to implement messaging functionality and transformation logic [Cha04]. An Enterprise Service Bus integrates numerous applications together over a bus-like infrastructure and enables applications using a multi-protocol message bus to communicate with each other. It incorporates the features required to implement a Service-Oriented Architecture and supports hiding complexity, simplifying access, allowing developer to use generic, canonical forms of query, access and interaction, handling the complex details in the background. An ESB should provide at least seven core functionalities, as following [RD08] and [Cha04]:

- Location transparent
  The ESB decouples the service consumer from the service provider. This means that a service consumer communicates with a service provider (or an application) without awareness of actual location of the service provider, even if there is a change in the location.

- Transport protocol conversion
  A service consumer and a service provider usually use different transport protocols. This leads to conflicts during their communications. An ESB should have the capability to convert incoming transport protocols to different outgoing transport protocols, like HTTP(S) to JMS or SMTP to TCP.

- Message transformation
  An ESB should be able to adapt the incoming messages into the format that the target application can actually accept.

- Message routing
  Almost all integration implementations should have message routing functionality. An ESB has to determine the ultimate destination of a particular incoming message, especially in the case that the message must be sent to multiple target applications.

- Message enhancement
  The destination applications may need more data other than the data provided by their incoming message. An ESB should enrich the incoming message with additional data that are retrieved from (external) data resources. Message enhancement is closely related to message transformation. The main difference is whether the existing data in the incoming message are enough to produce the correct outgoing message.

- Security
  A business-critical logic often involves a mass of applications. An ESB must provide authentication, authorization and encryption mechanisms to prevent malicious use of the ESB and satisfy the security requirements of service provider.

- Monitoring and management
  In order to ensure high performance and reliability of an ESB and to monitor the runtime execution of the message flows, a monitoring and management environment is necessary.

This is not a complete list of the functionalities of an ESB. Other functionalities, e.g. orchestration and transaction handing, are not mentioned. Here we present these seven important functionalities just to give a preliminary impression about ESB.

2.4 Java Business Integration

An Enterprise Service Bus acts as an integration middleware for Service-Oriented Architecture. However, implementation of an ESB is vendor-specific. Each vendor could use various protocols and technologies to design his own service container and interfaces for connecting and integrating services.

Java Business Integration (JBI) is a Java Specification Request 208 (JSR 208) that was developed under the Java Community Process (JCP). It defines an integration solution based on SOA concepts. All applications considered as loosely coupled function units are deployed into JBI components in the JBI environment. The JBI environment is responsible to route message exchange among the JBI components [Cor12]. JBI constructs a service container as a part of an ESB to provide a standardized, plug-in architecture for services integration. Several open-source projects have employed this approach, such as Open ESB [Bus], Apache ServiceMix [Fouc], Mule ESB [Com] and Fuse ESB [Cor12]. The JBI Runtime Environment manages JBI components and a Normalized Message Router (NMR).

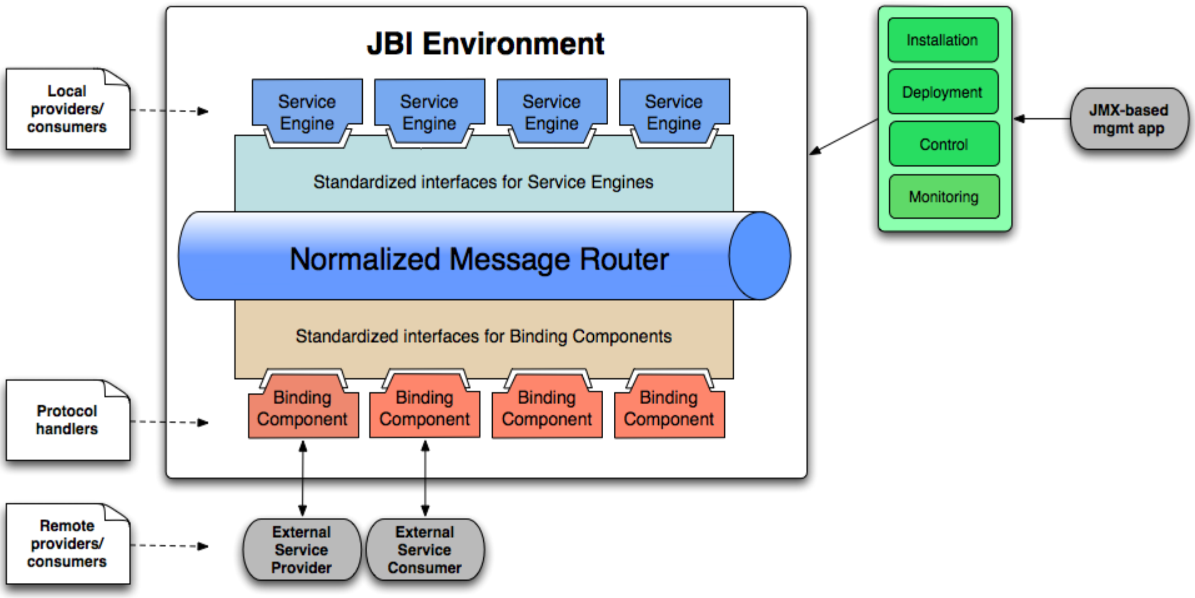

JBI Environment
Local providers/ consumers
Protocol handlers
Remote providers/ consumers
Service Engine
Service Engine
Service Engine
Service Engine
Standardized interfaces for Service Engines
Normalized Message Router
Standardized interfaces for Binding Components
Binding Component
Binding Component
Binding Component
Binding Component
External Service Provider
External Service Consumer
Installation
Deployment
Control
Monitoring
JMX-based mgmt app

Figure 2.2: Overview of JBI architecture (refers to Figure 1.1 in [Cor12])

In order to understand the JBI environment black-box, we should firstly understand how a component interacts with it (see Figure 2.2). There are two kinds of components that can be plugged into a JBI environment [Cor12] and [THW05]. The first one is Service Engine (SE). It provides business logic to other components, for example: message transformation, orchestration, advanced message routing. A Service Engine is seen as a container of an ESB to deploy functional units and can communicate only with the components locating inside the JBI component. The other one is Binding Component (BC). It acts as a connector using various communication protocols (SOAP, HTTP, File . . .) to access services which are outside the JBI environment. The messages from external sources should be normalized by the Binding Components, after they enter the JBI environment and before they are passed to the Normalized Message Router. The BCs are also responsible for transforming these Normalized Messages (NM) into the appropriate format of the destination services. These two components enable that business logic and integration logic are explicitly decoupled.

The Normalized Message Router is responsible for the message exchange between JBI components. It routes the Normalized Messages (normalized by Binding Components) from source JBI components to the correct destination JBI components. This enables fully decoupling the components and functionalities they expose.

JBI defines a common packaging model for deploying artifacts into the JBI environment. The two most often used types are Service Units and Service Assemblies. Service unit contains the functionality to be deployed into a JBI component. Service assembly is a collection of service units. Both are packaged as ZIP archives.

In general, JBI is salable and extensible. It facilitates integration of services and communication of service provider and consumers.

2.5 OSGi (Open Service Gateway initiative)

An OGSi framework [Bar09] specifies a modular system for Java. It enables applications to be modularized and resources to be shared between components within a single Java Virtual Machine (JVM). The central idea of OGSi is that each module owns its classpath separated from the classpath of all other modules to avoid the global, flat classpath which could cause many problems in traditional Java. Meanwhile, OSGi provides well-defined rules with a mechanism of explicit imports and exports to share classes across modules [Bar09]. In OSGi, modules are called bundles and packaged as JAR files. In addition to Java class files, metadata is placed in the *MANIFEST.MF* file inside the JAR file. Metadata explains what kinds of Java packages it provides (capabilities) and what kinds of Java packages it requires (requirements). After a bundle is installed, the requirements of the bundle would be first resolved and it would search for the matching capabilities offered by other installed bundles. Therefore, a bundle can either provide classes and services for others or implement its functionalities.

OSGi has defined a life cycle of a bundle that includes installed, resolved, starting, active, stopping, and uninstalled [All11]. The framework enforces the transitions between states. The first state is "installing". Having a bundle installed to an OSGi framework, its requirements as well as dependencies have to be immediately "resolved" instead of directly being started. If all the dependencies of this bundle are met, the bundle is ready to be started. "Starting" is a temporary state. The resolved bundle goes quickly into a running "active" state; meanwhile it provides or consumes services in the OSGi environment. Through transitory "stopping" state, the bundle is stopped and its state is transferred from "active" state to "resolved" state. Finally, the bundle could be removed namely "uninstalled" from OSGi framework.

2.6 Apache ServiceMix

Apache Software Foundation has proposed an open-source Enterprise Service Bus named Apache ServiceMix. Apache ServiceMix combines the functionality provided by a SOA model and the modularity provided by OSGi framework to decouple the applications and reduce their dependencies. It was built on the JBI specification JSR 208 allowing components and services to be integrated in a vendor independent way [Fouc].

Specifically, Apache ServiceMix unifies a number of existing solutions like Apache Karaf, Apache ActiveMQ, Apache Camel, etc. to provide a runtime platform for building integration solutions. Apache Karaf is the kernel layer of ServiceMix. It is an OSGi-based runtime and provides a container to manage the lifecycle of components, such as OSGi bundles, JBI components or service assemblies. Users can copy the JAR files of OSGi bundles into *$home/deploy* directory for deployment and delete them from the directory for undeployment. This feature is called hot-deployment [Fou11b]. Apache ActiveMQ embedded in ServiceMix is a JMS message broker.

It cooperates with Apache Camel to provide easy-to-use message persistence and reliable messaging [Foua]. ServiceMix employs ActiveMQ as foundation of the NMR that is responsible for routing the messages between the endpoints. The message routing among BCs, SEs and NMR has been explained in Section 2.4 (see Figure 2.2). The Apache Camel is a powerful open-source integration framework based on known Enterprise Integration Patterns. It enables routes to be built for integration of components quickly and efficiently [Fou11a]. The logic endpoints between Binding Components and the routing paths between the endpoints can be configured by deploying their configurations in service assemblies. The Apache Maven plugins simplifies the development process of JBI components and service assemblies [Foub].

2.7 ESB^{MT} and JBIMulti2

Multi-tenancy is an essential property of Cloud computing. It enables a single system instance to serve multiple customers by sharing computational resources between them. Therefore, Cloud service providers are able to best utilize their resources and reduce their servicing costs for each customer in order to maximize the profits. However, multi-tenancy transparency has to be guaranteed. That means Cloud service consumers should have the impression that they are the only one who uses the applications. The resource sharing must have no impact on application performance.

[SAL$^+$12] has defined multi-tenancy as "the sharing of the whole technological stack (hardware, operating system, middleware and application instances) at same time by different tenants and their corresponding users". Tenants and users are two types of consumers for multi-tenancy. Tenants are the consumers that are considered as groups like companies, organizations or departments. Users are the consumers that potentially belong to more than one tenant.

[SAL$^+$12] and [SAGSL13] focused on the PaaS model and investigated how its key component like the ESB could be multi-tenant aware, without caring about the particular implementation technology that the ESB uses. Based on the case study of 4CaaSt project [4Ca], a set of functional and non-functional requirements for multi-tenant ESBs have been identified [SAL$^+$12]. A multi-tenant ESB must offer the following functionalities. Firstly, an ESB must support managing and identifying multiple tenants. Secondly, the components of ESBs and the services for a certain tenant should be deployed, configured and managed in a transparent manner. Thirdly, each tenant could employ own interface for administration and management of tenants or users. Fourthly, a shared registry of tenants/users and a shared registry of services must be provided for the ESB and other PaaS components. Lastly, the ESB must be backward compatible with non multi-tenant services and applications. Apart from the required functionalities, several non-functional properties must be considered, such as tenant isolation, security, reusability and extensibility. Data isolation and performance isolation must be ensured between tenants. The ESB should provide authorization, authentication, integrity and confidentiality mechanism for security. The multi-tenant ESB should not be solution-specific and based on certain technologies, its components could be extensible and reusable by other PaaS's components when required.

Tenant awareness is a critical functional requirement. The concept Tenant Context has been introduced to enable multi-tenant aware messaging [SAGSL13]. A Tenant Context is represented in a structured format and contains two parts namely a mandatory part and an optional part. The combination of *tenantID* and *userID* which construct the mandatory part uniquely identifies a Tenant Context. The optional part consists of additional information like the name of the tenant and is represented as key-value pair to guarantee its extensibility. Furthermore, the Tenant Context should support both messaging protocols that integrate structure information into messages metadata, as well as ones that do not support such structured metadata through adding the Tenant Context in Extensible Markup Language (XML) format as string in the message metadata. Therefore, the Tenant Context could support various communication protocols and enable multi-tenant aware messaging to be independent of the technologies or protocols uses.

A message processing cycles of ESBs consists of Receive Message phase, Demarshalling phase, Process Message phase, Marshalling phase and Send Message phase [SAGSL13]. In the Receive Message phase, the ESB receives a communication protocol-specific message. In the Demarshalling phase, the message is mapped and transformed into an internal Normalized Message (NM). The NM thus contains the information of the Tenant Context which is integrated into the metadata of the original message. Then, the internal business logic of the ESB could access the required information from the multi-tenant aware NM. After the processing there is the Marshalling phase in which the NM that contains the results could be mapped and transformed into a format that the recipient of the message needs. In the final Send Message phase, the resulting message is sent to its target.

A generic multi-tenant ESB architecture (ESB^{MT}) has three layers: a Presentation layer, a Business layer and a Resource layer [SAL+12]. It will be introduced in a bottom-up fashion and referred to the work of Dominik Muhler [Muh12].

Dominik Muhler has developed a system named JBIMulti2 which is the abbreviation of "JBI multi-tenant multi-container Support". It supports a multi-tenant management application to control a set of databases and clusters of multiple JBI containers, which ensure the elasticity characteristic of cloud computing [Muh12]. This system is based on Apache ServiceMix 4.3.0 that implements the JBI specification.

It can be indicated from Figure 2.3, that the Resource layer located at the bottom consists of an ESB Instance Cluster and a set of registries. Each ESB Instance executes the functionalities of a traditional ESB solution, which were discussed in Section 2.3. In order to fulfill the functional and non-functional requirements introduced previously, an ESB should be extended to be multi-tenant aware. For example, all components can process messages that include tenant and user information. The ESB could route messages between endpoints according to the tenant information in the messages. The ESB must prevent tenants and users from accessing the messages of other tenants and users. Furthermore, all components of the ESB have to be tenant- and user-specific configurable. This means that each tenant could have its own endpoint for communication. The backward compatibility for non multi-tenant components must be ensured.

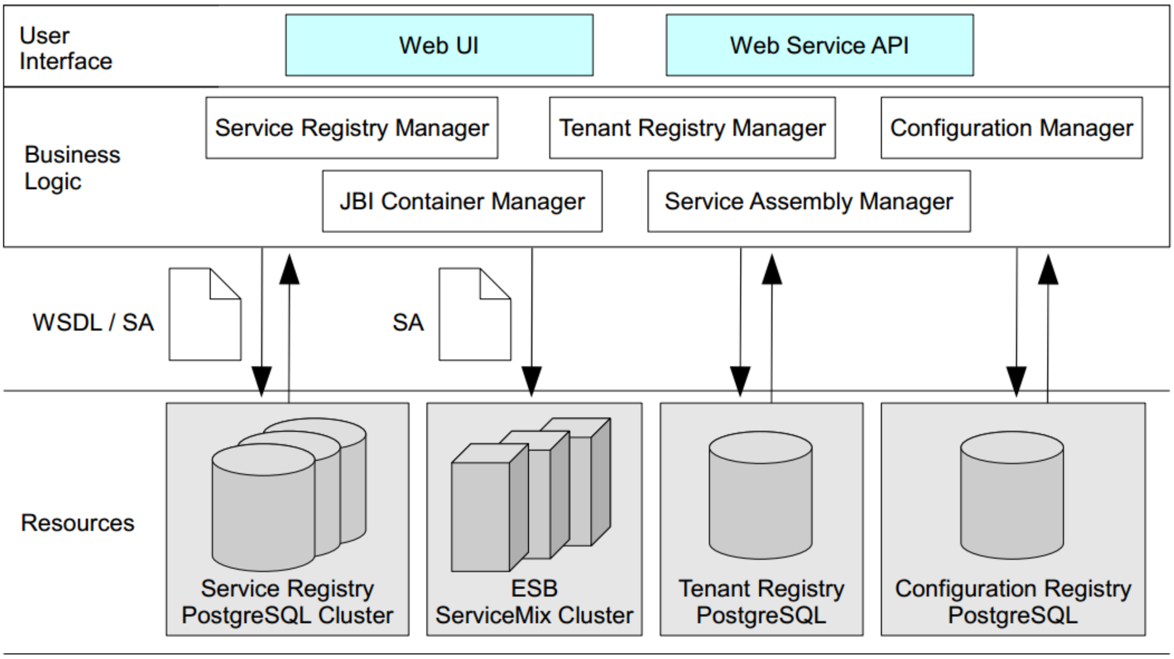


Figure 2.3: Overview of JBIMulti2 [Muh12]

Additionally, there are three types of registries in the Resource layer namely Tenant Registry, Service Registry and Configuration Registry. They were implemented using platform-wide PostgreSQL 9.1.1 databases [SAL+12].

- The Tenant Registry keeps tenant context for tenant users and their roles inside corresponding tenant. Tenant users belong only to one tenant. Every tenant and tenant user is identified by a Universally Unique Identifier (UUID) and can have associated properties as key-value pair.
- The Service Registry stores service assemblies and service descriptions in a tenant-isolated manner for each tenant. Service assemblies are saved as ZIP files, while services are saved as the WSDL description in XML documents.
- The Configuration Registry stores all other non-tenant related and tenant related data.

In the Business Logic layer, the functionality for tenant awareness and security is encapsulated in an Access Layer component. Access Layer acts as a multi-tenancy enablement layer and implements role-based access control function using interceptors [Muh12]. The roles are classified into system administrators, tenant administrators and tenant operators. A system administrator does not belong to any tenant. He configures the system and assigns quotas resources usages to the tenants. This administrator has unlimited permissions and can interfere in the actions of the tenant users. Tenant administrators define roles and assign permissions to them. A tenant administrator can manage contingents of service units and service registrations. He continuously partitions the quota of resource usage assigned by the system administrator.

Tenant operators can access the resources using a resource contingent assigned by the tenant administrator. Before an execution of a business method, such interceptors are called to check, if the caller is authenticated and has permission to execute the current business method that is annotated with the required permission type. The interceptor can check the authentication via comparison of this extracted information with the information retrieved from the Tenant Registry and the Configuration Registry.

Furthermore, Tenant Registry, Configuration Registry and Service Registry Managers in the Business Logic layer are responsible for managing and executing the business logic to respective registries in the Resources layer (see Figure 2.3). All operations to underlying data resources must ensure consistency by distributed transactions. JBIMulti2 web applications use container-managed transaction demarcation for this purpose. Management operations to JBI containers are sent by *JBIContainerManager* to a messaging topic. All JBI containers are selective, transactional and durable subscribers. *JMSManagementService* on container side receives the subscribed messages. This Publish-Subscribe pattern guarantees reliable messaging. Additionally, JBIMulti2 uses *JMSManagementService* to extend the JBI components on each ServiceMix instance by adding the tenant context as XML documents to each service unit contained in a service assembly. Once they are deployed on ServiceMix instances, they do not interfere with service units of other tenants.

The Presentation layer is built on the top of the Business Logic layer. It consists of the Web UI and the Web Service API, in order to customize, manage and interact with a multi-tenant aware ESB.

[Muh12] has developed a multi-tenant enterprise application that allows tenant users to deploy JBI service assemblies to JBI containers of Apache ServiceMix. Based on the role-based access control, system administrators, tenant administrators and tenant operators are distinguished in their permissions. [Muh12] has extended additionally the Apache ServiceMix for communication with enterprise application and ensuring the data isolation of deployed JBI artifacts. To gain a deep understanding of JBIMulti2 was the first task, because our work is based on this system.

2.8 Service Evolution

Services have to be continuously evolved to fulfill the increasing requirements for competition and innovation. Service evolution has been defined as "the continuous process of development of a service through a series of consistent and unambiguous changes" [Pap08]. Each service developer should know why a change was made, which implications it has, and whether it is consistent, so that the service evolution could be carried out smoothly and steadily.

Service changes could be classified as shallow changes or deep changes [PAB11]. Shallow changes are small-scale incremental changes. They are normally localized to a service and/or restricted to the clients of that service. Shallow changes typically contain structural changes (like service types, messages, interfaces, and operations) and business protocol changes. On the other hand, deep changes are large-scale transformational changes which extend beyond the consumers of a service, possibly to all consumers of an entire end-to-end service chain. Deep changes are operational behavior changes, non-functional changes, and policy induced changes.

We will further focus on shallow changes and discuss service versioning and compatibility based on the structural service changes.

2.8.1 Service Versioning

Versioning is a concept for software maintenance and evolution which originates from the Software Configuration Management (SCM) field [BR00]. Once software artifacts have changes, a historical record has to be kept. Service versioning has two dimensions: interface versioning which supports the service description, and implementation versioning which supports the code, resources, configuration files and documentation of a service. In follows only interface versioning will be handled.

Service version is usually named as Major#.Minor#, a naming scheme which consists of a major release version number and a minor release version number. Major release means breaking changes, while Minor release represents non-breaking changes [JD08]. For instance a service with version number "3.2" denotes the second minor version of the third major release. As for another naming approach, date stamp is used to identify each service version [PAS04].

XML Schema provides the mechanisms for versioning of Web Service [ABP12]. Introducing new XML namespaces to either service itself or its data types could cause breaking the binding to the service on the consumer side. Therefore, this method is only used to create a major version of a service. Moreover, as attributes in the elements document or as part of the URL, Version Identifiers (VIDs) could be used to unambiguously name a service version. Both methods are based on the naming scheme Major#.Minor# and can be combined for version control. As alternative or complementary to XML-based approaches, service registries like UUDI or custom registries which store and control the version information could be used for service interface versioning. For this purpose, the versioning data need to be added into service description model where the registry works.

A common strategy for versioning is compatibility-oriented [ABP12]. The developers maintain multiple active service versions for major releases at same time. Despite the grouping of all minor releases into the latest major release for reducing maintenance costs, the cost still varies in proportion to the number of active versions. The creation of a major version could therefore result in breaks of existing consumers as well as an increase in maintain cost.

2.8.2 Service Version Compatibility

Service version compatibility is an essential concept in service evolution and is closely referred to service versioning. It guarantees that a newly emerging service version of either provider or consumer of service messages cannot impact on each other. It has usually two cases [PAB11].

- Backward compatibility
  The introduction of a new version on the message consumer side cannot affect the message providers. The new service version should continuous support the old versions.
- Forward compatibility
  A new version of a message provider is introduced without the impact on the message consumers. The new service version should continuous support the old versions.
- Full compatibility
  If a new service version fulfills both conditions of backward compatibility and forward compatibility, it is fully compatible.

In practice, the factor that distinguishes a minor release from a major release is backward compatibility [ABP12]. A new service version is regarded as a minor release if its changes are backward compatible; otherwise, if the changes are backward incompatible, it is called major release.

Change	Backwards Compatible
Add (Optional) Message Data Types	Yes (input only)
Add (New) Operation	Yes
Modify Service Implementation[a]	Yes
Remove Operation	No
Modify Operation[b]	No
Modify Message Data Types	No
a: As long as it has no effect on the service interfaces. b: Includes renaming, changing parameters, parameter order, and message exchange pattern.	

Table 2.1: Guidelines for Backward Compatible Changes [ABP12]

Table 2.1 lists guidelines for compatibility assessment. All changes are represented as changes to WSDL and XML schema elements. Obviously, the addition of elements related to input data types or operations, or the modification of service implementation which has not effects on the WSDL document is backward compatible changes. The removal or modification of an operation can lead to breaking the message providers and is therefore strictly prohibited. The compatibility assessment approach based on these guidelines is widely accepted because of its usability and minimum demand for infrastructure.

Although widely used, the guideline-based approach has obvious disadvantages. For example, the compatibility between two services is deduced by service developers. This subjective deduction and the lack of a solid theoretical foundation could lead to errors. Thus, a Compatibility Checking Function has been proposed by [ABP12] for objective checking the compatibility of service versions. The implementation of the CCF algorithm is another main task of this thesis.

Previously, the service version compatibility has been classified into backward compatibility, forward compatibility and full compatibility. From another aspect it can be classified into horizontal compatibility and vertical compatibility [And10].

- Horizontal compatibility or interoperability
  Two services can successfully interact with each other, either as service providers or service consumers.
- Vertical compatibility or substitutability (from the provider's perspective) or replacement (from the consumer's perspective)
  One service version could replace another service version in a given context.

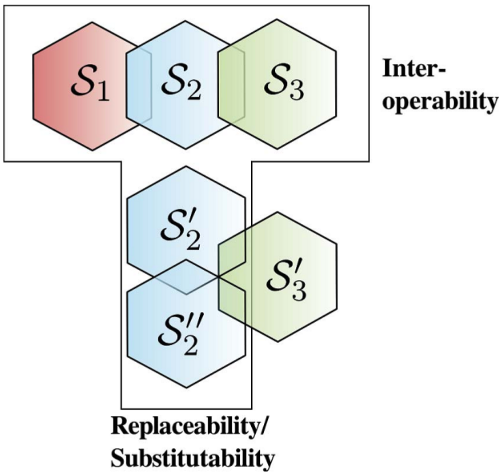


Figure 2.4: Horizontal and vertical compatibility [ABP12]

These two dimensions of service version compatibility construct a notation of T-shaped changes. The compatible service versions are represented by the overlapping parts of the hexagons. The Figure 2.4 shows that service S_1 is horizontally compatible with service S_2, and S_2 is horizontally compatible with S_3. Service S_2 has additionally two versions S'_2 and S''_2 that are vertically compatible with each other. The gap between two services stands for the incompatibility of them, such as S_2 and S'_2. In other words, if S_2 is a major release, then S'_2 will be another major release (because their interoperability is broken). Service S''_2 will be a minor release of S'_2 and it can therefore replace Service S'_2.

In order to formally define the service compatibility between any two services, the Abstract Service Description (ASD) meta-model is introduced. The ASD meta-model has three layers. The structural layer includes an Operation and a Message concept which respectively relate to the WSDL operation and message constructs. The behavioral layer describes the how services perform once a connection between service consumer and service provider is established. The non-functional layer uses e.g. a set of policy constraints of assertions to ensure Quality of Service. In this thesis, only the structural layer will be considered.

A description schema S expresses a service version and consists of a set of records s that describe the structural dependence inside the service description using elements and their relationships [ABP08]. Besides, a subtyping relation is introduced to (partially) order records for determining the compatibility. If a record s is a subtype of a record s', it can be represented as $s \leq s'$.

S is able to be divided into two subsets S_{pro} and S_{req}. S_{pro} consists of output-type records of a service description, while S_{req} consist of input-type records. The compatibility between two service versions S and S' can be formally defined as follows [ABP12].

- Forward compatibility
  $S <_f S' \Leftrightarrow \forall s \in S_{pro}, \exists s' \in S'_{pro}, s' \leq s$ (covariance of output).
- Backward compatibility
  $S <_b S' \Leftrightarrow \forall s' \in S'_{req}, \exists s \in S_{req}, s \leq s'$ (contravariance of input).
- Full compatibility
  $S <_c S' \Leftrightarrow S <_f S' \wedge S <_b S'$.

The term T-shaped change relates to the horizontal and vertical compatibilities. A change set $\triangle S$ which will take place at a service description S, is a T-shaped if and only if it leads to a fully compatible service description S' (formally, $S' = S \circ \triangle S$ and $S <_c S'$). Using the type theory for the structural layer of an ASD and subtyping relation of ASD records, the assessment for compatibility of service versions becomes possible. A Compatibility Checking Function algorithm has been proposed by [ABP12].

Algorithm 2.1 Compatibility Checking Function (CCF) algorithm [ABP12]

```
for all $s' \in S'_{req}$ do
  if $\nexists s \in S_{req}, s \leq s'$ then
    return false
  end if
end for
for all $s \in S_{pro}$ do
  if $\nexists s' \in S'_{pro}, s \leq s'$ then
    return false
  end if
end for
return true
```

Algorithm 2.1 shows the CCF algorithm for checking the compatibility between service descriptions S and S'. In this case, the service description S is divided into S_{pro} and S_{req} as well as S' into S'_{pro} and S'_{req}. Records s belongs to service description S, while s' belongs to S'. The first iteration (from line 1 to line 5) and the second iteration (from line 6 to line 10) evaluate the backward compatibility for S'_{req} and the forward compatibility for S'_{pro}, respectively. If both iterations return *true*, that indicates the new service version S' is fully compatible with the previous service version S.

2.8.3 Identification Model of Service Change

Three main models have been summarized by [ABP12] to perceive and identify service changes for service consumers. The first model is called client model [BE04], [JD08] and [JSBR09] in which a new service version caused by both nonbreaking and breaking changes has to be recognized directly by service consumers. This can be achieved for example by checking the service registry regularly. The second model is notification model [FLF+07] in which a service consumer will receive a notification once a new service version is released. The last one is transparent model. A new service version with nonbreaking changes does not need to be identified by consumers. These consumers could continuously use the existing service without any problems [NS07].

2.9 The Extension of the JBIMulti2 management system for service version management

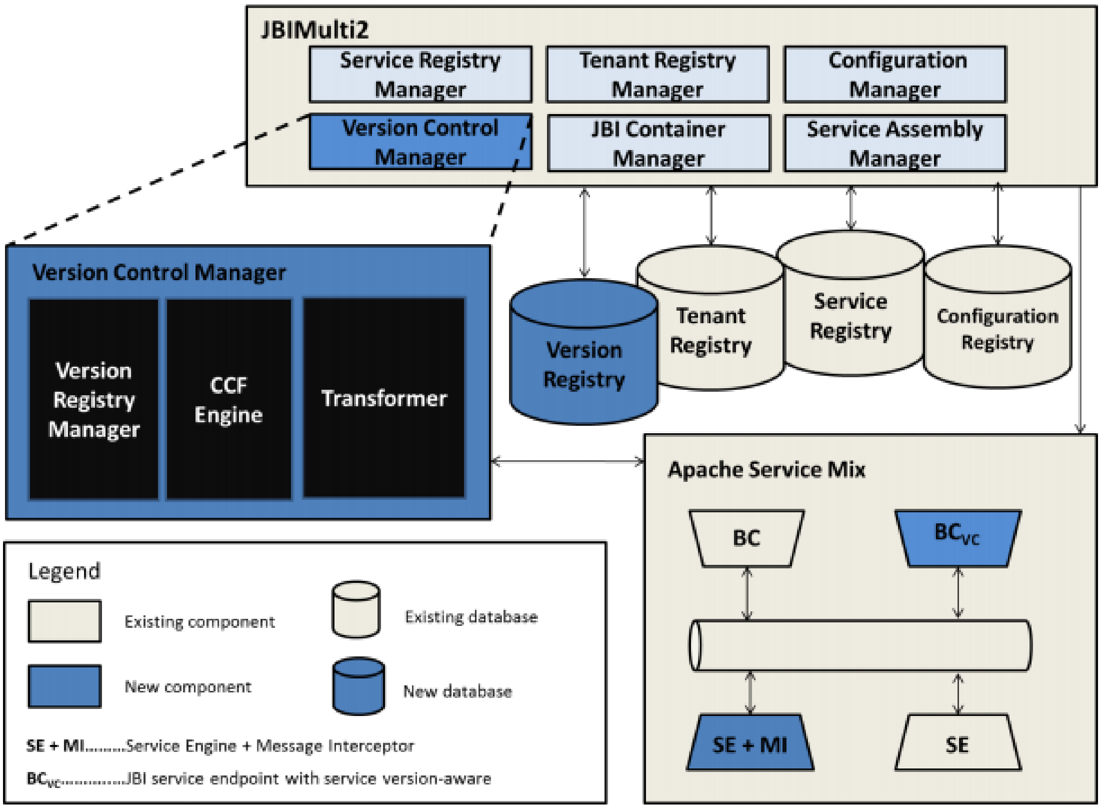


Figure 2.5: Service Version Control Management system by [Lie13]

The previous work from Sumadi Lie has extended the JBIMulti2 system with a service version control management system which manages all service interface description registered by service providers [Lie13]. This system consists of three main components, namely the Version Registry, the Version Control Manager and the multi-tenant aware ESB (see Figure 2.5).

The Version Registry has been designed to keep all service versioning-related information and history. The Version Control Manager enables the Version Registry to communicate with the JBIMuiti2 system as well as executing operations on the Version Registry.

The Version Control Manager has three main components: the Version Registry Manager, the CCF (Compatibility Checking Function) Engine and the Transformer. The Version Registry Manager listens to the events of operations on the Service Registry and then calls on the CCF Engine to run a compatibility assessment between the new service version and the old service version if necessary, so that the Version Registry can be updated in time. The Transformer is responsible to transform an incoming request message to a corresponding outgoing message and vice versa, if the incoming message is compatible with the service version which is implemented by the service endpoint.

Sumadi Lie has attempted to make Binding Components service version aware by embedding unique service version identifiers into endpoint names, so that incoming messages targeted at these endpoints could be routed by the ESB. Additionally, he has proposed integrating Message Interceptors into Service Engines which have routing capability. These Interceptors listen to the communication channel. When a message is compatible with the Service Engine, the Interceptor in the SE will trigger the Transformer.

Although [Lie13] has conceived a service version management system, its implementation has remained partial. The CCF assessment has also not been realized. Moreover, in [Lie13], the author has assumed that the replacement operation would be a sequential execution of an unregistration operation and a registration operation, and a new service interface for this new service would be created. In practice however, instead of creating a new service interface, the old service interface will be retained and assigned to the new service version. Therefore, the extension of JBIMulti2 with service version management should be further improved and implemented in this work. The reasonable suggestions from [Lie13] would help us to achieve our goals.

3 Specification and Design

In this chapter the specification of the service version control management system is described and the concept of the system design is explained. In the first section, we will discuss the requirements for this system. In the second section, three use cases are constructed to describe the functionalities of the service version control management system. In the third section the concrete design will be expounded. Finally, we summarize this chapter briefly.

3.1 Requirements

Services have to be continuously evolved (see Section 2.8). This might result in existence of multiple versions of a service. Either compatible or incompatible versions are probably provided to various consumers at the same time (see Figure 3.1). Thus a system for managing multiple versions of a service is necessary. Although the JBIMulti2 system implements simple functionalities for service management, like service registration and unregistration, it cannot deal with multiple versions for a particular service. Therefore the JBIMulti2 system is required to be extended for service version management.

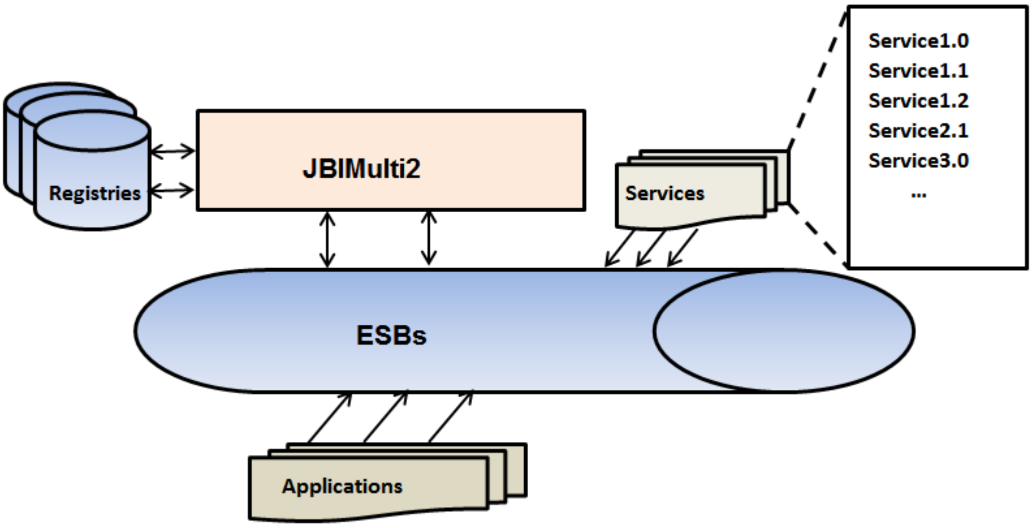


Figure 3.1: Service version management

This service version control management system will be designed only for maintaining all service interface descriptions. In order to completely understand the functionalities of this system, several questions related to service versioning must be considered.

- Which service interface descriptions will be managed and versioned?
  In general, all service descriptions should be managed and versioned. They should be registered within a system and exposed as service endpoints.
- When service compatibility between service versions will be checked?
  Typically, the compatibility assessment is executed after a new service has been registered and at least two services have been available. It happens, for instance, when a service is replaced by another.
- Where the service interface versioning information is stored?
  The service versioning information is registered by a Version Registry.
- How the service versions are uniquely identified?
  Service providers are not in charge of service version numbering. In addition, the service description does not contain any version-related information. The version number could be determined based on the result of compatibility assessment and the related service version.

3.2 System Specification

All operations of JBIMulti2 management system are based on role-based control [SCFY96]. Each role has to access the system and data resources with appropriate permissions. An operation beyond its privileges is forbidden. There are three main roles in the JBIMulti2 system, namely, system administrator, tenant administrator and tenant operator, as discussed in Section 2.7.

In order to show how the JBIMulti2 management system for service version control management is extended, three use cases are introduced: to register a new service, to replace service and to deploy service in parallel as a system administrator. The extended use case diagram of the JBIMulti2 system is shown in Figure 3.2 by which the new use cases are sketched as gray ovals. Detailed descriptions for these use cases can be found in Table 3.1, Table 3.2 and Table 3.3, respectively.

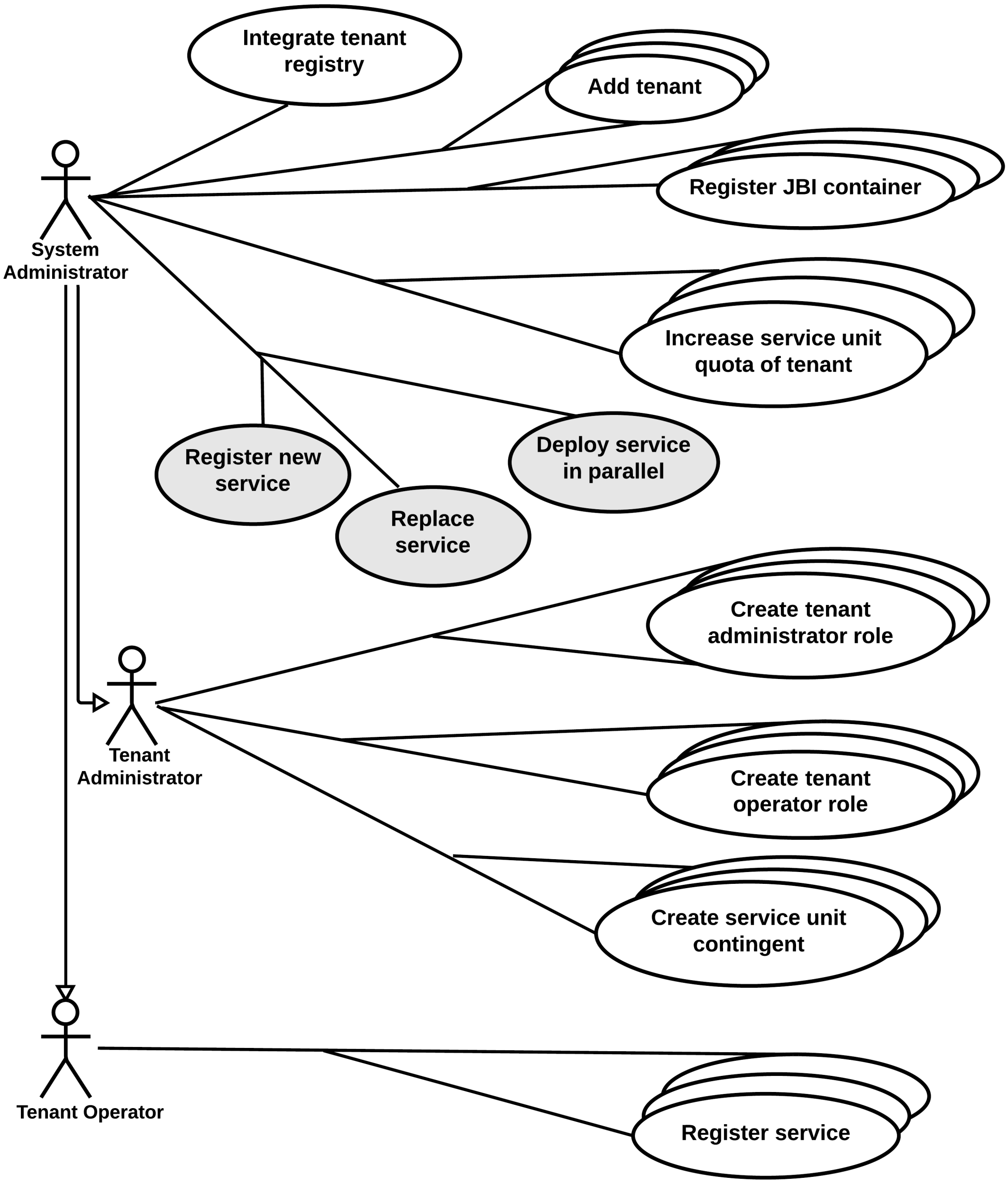

Integrate tenant registry
Add tenant
Register JBI container
Increase service unit quota of tenant
Register new service
Replace service
Deploy service in parallel
System Administrator
Create tenant administrator role
Create tenant operator role
Create service unit contingent
Tenant Administrator
Tenant Operator
Register service

Figure 3.2: Use Case Diagram extension based on JBIMulti2 (extending [Muh12])

Name	**Register New Service**
Goal	The system administrator wants to register a new service with version number 1.0.
Actor	System Administrator
Pre-Condition	There is no an active service version in the Version Registry which has the same service name with the new service.
Post-Condition	The new service is registered in Service Registry and the version-related information is stored in the Version Registry.
Post-Condition in Special Case	The new service is not registered. No versioning information is added into the Version Registry.
Normal Case	1. The system administrator commands the system to register the new service. 2. The new service is registered by adding *versionedServiceName* (namely serviceName#1.0) and WSDL document into the Service Registry. 3. The service versioning information including setting the *serviceStatus* as active is stored in the Version Registry. 4. The system finishes the transactions.
Special Cases	1. An active service version has already existed in the Version Registry which has the same service name with this new service. - The system shows a notification: "This service has existed already, please try *replaceService* or *deployParallel* operation!" and aborts. 2. The system cannot finish the transaction with the Service Registry. - The system rolls back the transaction and shows an error message. 3. The system cannot finish the transaction with the Version Registry. - The system rolls back the transaction and shows an error message. 4. The service description document is not a valid WSDL file. - The system shows an error message and aborts.

Table 3.1: Description of Use Case Register New Service

Name	**Replace Service**
Goal	The system administrator wants to replace an old service version with a new service version.
Actor	System Administrator
Pre-Condition	This old service does exist in the Service Registry. The new service version has to be compatible with the old service version.
Post-Condition	The old service version is unregistered from the Service Registry. The service versioning information about the old version is updated. The new service version is registered into the Service Registry. The service versioning information of the new version is added into the Version Registry.
Post-Condition in Special Case	The old service version is not unregistered and the new service version is not registered. The respective service versioning information cannot be updated.
Normal Case	1. The system administrator instructs the system to replace an old service version with a new service version. 2. CCF checks the compatibility between these two services and calculates the *serviceVersionId* of this new service. 3. The old service version is unregistered from the Service Registry, but its interface cannot be deleted. 4. Its service versioning information in the Version Registry is updated, such as *serviceStatus* is set as decommissioned. 5. The new service version is registered by adding the WSDL document and adding *versionedServiceName* (namely serviceName#serviceVersionId) into the Service Registry. Additionally, it reuses the interface of the old service version. 6. The service versioning information including setting the *serviceStatus* as active is stored in the Version Registry. 7. The compatibility status about of two services is set as "compatible". 8. The system finishes the transactions.
Special Cases	1. The old service does not exist in the Service Registry. - The system shows a notification: "This service to be replaced does not exist, please try *registerNewService* operation!" and aborts. 2. The new service version is incompatible with the old service version. - The system shows a notification: "This service is an incompatible version, please try *unregisterService* and *registerNewService*, or *deployParallel* operations!" and aborts. 3. The system cannot finish the transaction with the Service Registry. - The system rolls back the transaction and shows an error message. 4. The system cannot finish the transaction with the Version Registry. - The system rolls back the transaction and shows an error message. 5. The service description document is not a valid WSDL file. - The system shows an error message and aborts.

Table 3.2: Description of Use Case Replace Service

Name	**Deploy Service in Parallel**
Goal	The system administrator wants to deploy a new service in parallel with an existing service version.
Actor	System Administrator
Pre-Condition	The existing service does exist in the Service Registry.
Post-Condition	The new service version is registered and deployed in parallel with the old version. The service versioning information about this new service version is added into the Version Registry.
Post-Condition in Special Case	The new service version is not registered into the Service Registry. The Version Registry is not updated.
Normal Case	1. The System Administrator commands the system to deploy this new service version in parallel with an existing service version. 2. CCF checks the compatibility between these two services and calculates the *serviceVersionId* of this new service. 3. The new service version is registered by adding the WSDL document and adding *versionedServiceName* (namely serviceName#serviceVersionId) into the Service Registry. 4. The service versioning information including setting the *serviceStatus* as active is stored in the Version Registry. 5. The compatibility status of these two services is set as the result of the CCF assessment. 6. The system finishes the transactions.
Special Cases	1. The old service does not exist in Service Registry. - The system shows a notification: "This service version does not exist, please try *registerNewService* operation!" and aborts. 2. The system cannot finish the transaction with the Service Registry. - The system rolls back the transaction and shows an error message. 3. The system cannot finish the transaction with the Version Registry. - The system rolls back the transaction and shows an error message. 4. The service description document is not a valid WSDL file. - The system shows an error message and aborts.

Table 3.3: Description of Use Case Deploy Service in Parallel

3.3 System Design

This section explains the extension of the JBIMulti2 management system for managing service evolution in detail, which is the main subject of this diploma thesis. After a brief review, it is mainly discussed how the JBIMulti2 system is extended for service evolution management.

[Muh12] has designed and implemented a multi-tenant management and administration system based on Apache ServiceMix 4.3.0 that implements the JBI specification. This system has supported a cluster of instances of Apache ServiceMix for ensuing the elasticity characteristic of cloud computing. The most important is the data isolation between tenants, whether it relates to message flows inside the Enterprise Service Bus or service assemblies (see details in Section 2.7).

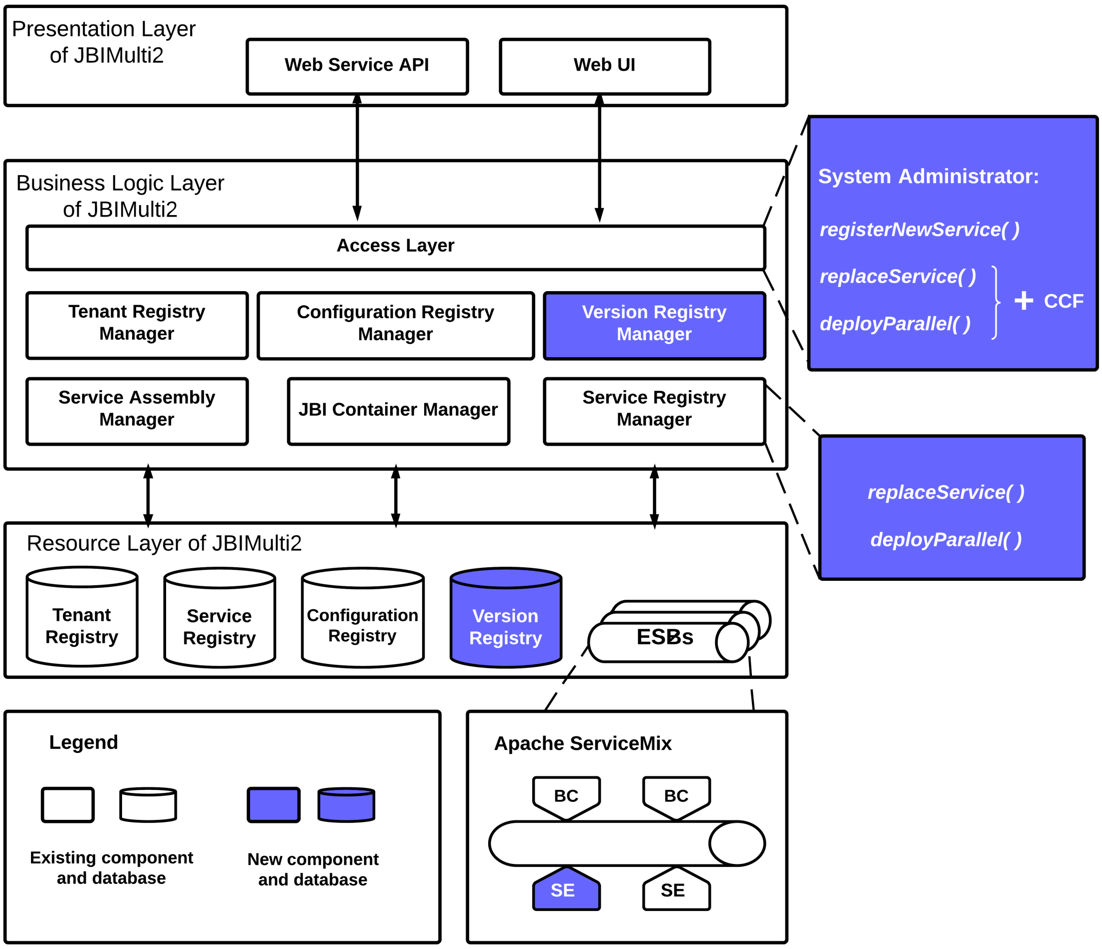


Figure 3.3: Overview of the service version control management system based on the system design by [Lie13]

The JBIMulti2 system has employed the following components for tenants and services to ensure the data isolation between tenants. A Service Registry is responsible for storing Service Assemblies and active services. A Tenant Registry stores information related to all tenants and their corresponding users. A Configuration Registry keeps configuration data. A Service Registry Manager registers service assemblies and services into the Service Registry. A Tenant Registry Manager stores and retrieves the tenants' information if needed. A Configuration Registry Manager performs storing configuration data into the Configuration Registry, which are created by system administrators or tenants. The Tenant Context can be added into each service unit inside a service assembly by a Service Assembly Manager, in order to make ESBs multi-tenant aware. A JBI Container Manager is used to interact with underlying multi-tenant aware ESB implementation.

Figure 3.3 sketches the extension of the JBIMulti2 system for service evolution management. The details of this extension will be explained in the following sections.

3.3.1 Modification and extension of JBIMulti2 components

In previous section we have introduced three use cases with the system administrator permission. These use cases combine with the CCF assessment to ensure the managing of service evolution. In order to achieve this goal, we extend the JBIMulti2 management system as follows:

- Access Layer
  The *SystemAdminFacadeBean* is enriched by adding the three functions *registerNewService*, *replaceService* and *deployParallel*. They provide the JBIMulti2 system with the capability to manage the service evolution.
  The CCF assessment is embedded into *replaceService* and *deployParallel*.
  In addition, a service description does not contain any versioning information. The *serviceVersionId* needed by the Version Registry has to be extra calculated based on the result of CCF assessment.

- Checking Compatibility Function (CCF)
  CCF is used to check the compatibility of services. Its algorithm has been discussed in Section 2.8.2. The implementation of CCF algorithm is another important task of this thesis.
  The result of CCF assessment could be considered either the value of compatibility status or the basis for calculating *serviceVersionId*.

- The Service Registry Manager
  ReplaceService and *deployParallel* are added as new functionalities of the Service Registry Manager.

- The Service Registry
  Because there are services which have identical service name, but different service version identifiers, the attribute *serviceName* of service entity in the Service Registry will be modified as *versionedServiceName* which consists of service name and service version identifier to uniquely define a service name.

- The Version Registry Manager
  It connects JBIMulti2 system and the Version Registry, and is responsible for updating the Version Registry, when a new service is registered, a service is replaced by a new service, or a service is deployed in parallel with an existing service.
- The Version Registry
  A Version Registry is used as a database to maintain service versioning information.
- A extended Service Engine
  A message interceptor is added into the Service Engine which is responsible for routing messages. A transformer could be added into the Service Engine or considered as a functionality of the service version control management system to adapt the message payloads into what the compatible service needs. These new components could ensure a dynamic routing of messages.

3.3.2 Database schema

3.3.2.1 The Version Registry

The JBIMulti2 management system does not have any databases to store the version-related information. Here a Version Registry is proposed for the managing of service interface versioning.

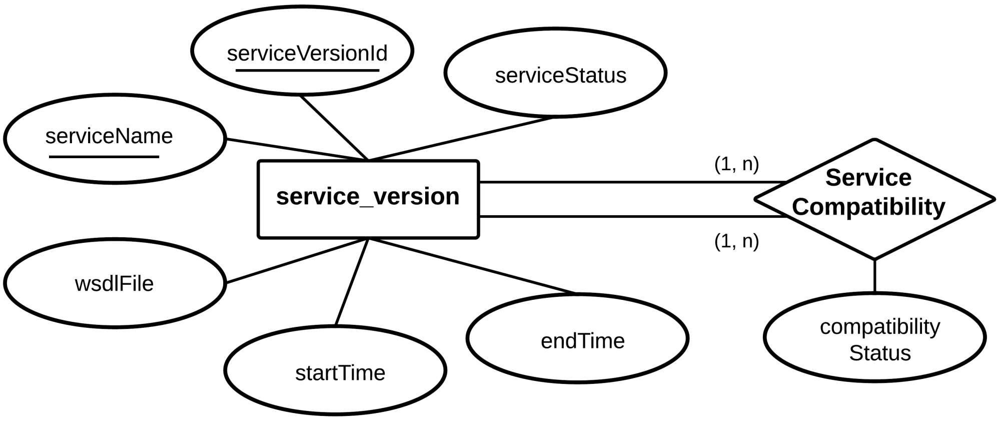


Figure 3.4: ER-Diagram of the Version Registry

Figure 3.4 is an Entity Relationship Diagram of the Version Registry which has two entities.

- Service version (*service_version*)
  Service version entity stores all versioning information for a service description. It contains the following attributes.
  - Service name (*serviceName*): the name of the service in WSDL.

  - Service version identifier (*serviceVersionId*): consists of a major release number and a minor release number, such as version 3.2.
  - Service status (*serviceStatus*): it would be set as active, if a service is successfully registered, while it would be set as decommissioned, if a service is unregistered.
  - Start time (*startTime*): indicates the time a service is set as active.
  - End time (*endTime*): denotes the time a service is decommissioned.
  - Service description (*wsdlFile*): the WSDL file of this service version.

- Service compatibility
  It represents a recursive relationship between service versions.
  - Compatibility status: describes whether two service versions are compatible. It could be set as "compatible" or "incompatible".

The service name combines with the service version identifier to uniquely identify a service version.

3.3.2.2 The Service Registry

The main parts of the Service Registry are not changed. Only the attribute service name (*serviceName*) of the service entity is replaced by a service name with its version number (*versionedServiceName*), so that a service could be distinguished from other services with the same service name by its version identifiers. The value of the *versionedServiceName* is in the form of serviceName#Major.Minor. For example, WeatherService#1.1 and WeatherService#1.2 could exist in the Service Registry simultaneously.

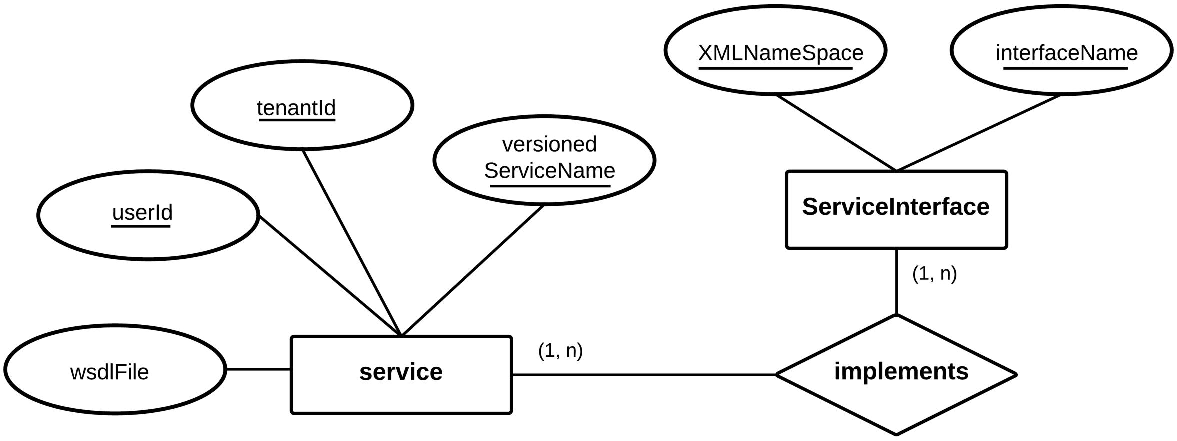


Figure 3.5: ER-Diagram of the Service Registry

Furthermore, when a new service is registered or a new service is deployed in parallel with an existing service, a new service interface will be assigned to this new service. However, when an old service version is replaced by a new service version, the service interface of the old service will be reused by the new service.

3.3.3 The Service Version Control Management

In Section 3.2, we have discussed three new use cases which allow system administrators to register new service, to replace an old service version with a new service version and to deploy a new service in parallel with an old service version. The implementation of these operations indicates that the extension of the JBIMulti2 system could provide version control capability. We will use several activity diagrams about use cases to show procedural executions and flows related to system components. Moreover it should be noted that the JBIMulti2 system has already provided the capabilities to register a service into the Service Registry by *registerService* and unregister a service from the Service Registry by *unregisterService*. But both of them cannot deal with the data in the Version Registry.

The first use case is sketched by Figure 3.6. The system administrator wants to register a new service as the first version 1.0. In this case, it has to be checked whether the Version Registry has already stored an active service version which has an identical service name. If such a service does exist, the system will show a notification "This service has existed already, please try *replaceService* or *deployParallel* operation!" and abort all transactions. If such a service does not exist, the new service could be registered into the Service Registry and its version-related information will be stored in the Version Registry. The service status is then set as active, means that this service version is the most actual version at that time. After the JBIMulti2 system has received an acknowledgment, all transactions of *registerNewService* are done.

Another use case is the replacement of an old service with a new service (see Figure 3.7). First of all, the existence of this old service has to be checked. If such service does not exist in the Service Registry, the system will display "This service to be replaced does not exist, please try *registerNewService* operation!" and abort all transactions. If such service does exist, the process could be continued. A service could be replaced only by a compatible service. Thus, CCF has to be executed to exam the compatibility of these two services. Furthermore, the version identifier should be determined based on the version of the old service and the result of CCF assessment. If all pre-conditions are fulfilled, the replacement can be performed. The JBIMulti2 system will first unregister the old service by removing it from the Service Registry and register the new one. Then the status of the old service in the Version Registry is set as decommissioned. All versioning information about the new service is added into the Version Registry and the status of the new service is set as active. Finally, the compatibility status for these two services is recorded as "compatible". After that, the replace service operation is finished. Note that the service interface of the old service is reused by the new service.

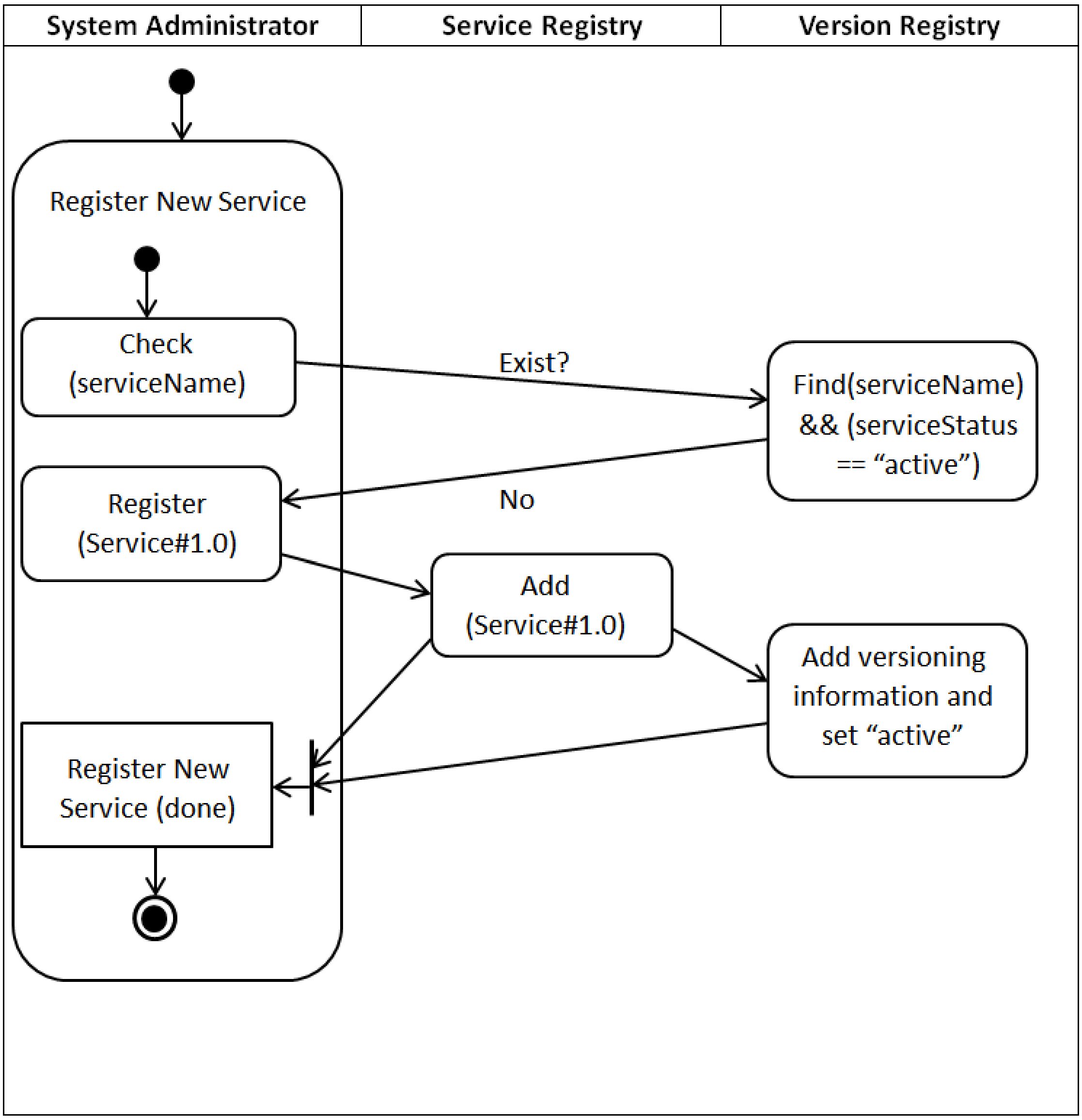


Figure 3.6: Activity diagram of register new service use case

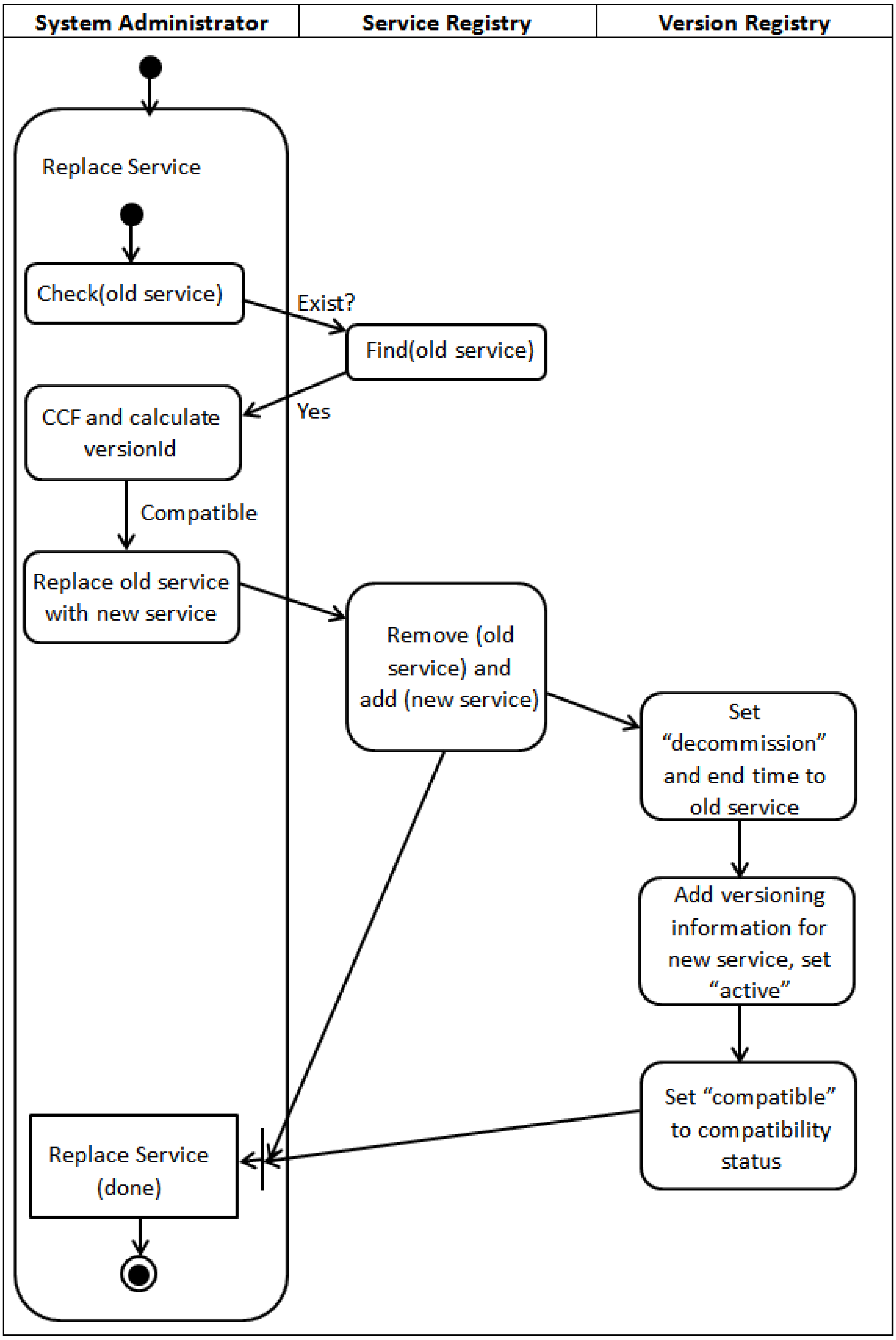

System Administrator
Service Registry
Version Registry
Replace Service
Check(old service)
Exist?
Find(old service)
Yes
CCF and calculate versionId
Compatible
Replace old service with new service
Remove (old service) and add (new service)
Set "decommission" and end time to old service
Add versioning information for new service, set "active"
Set "compatible" to compatibility status
Replace Service (done)

Figure 3.7: Activity diagram of replace service use case (compatible)

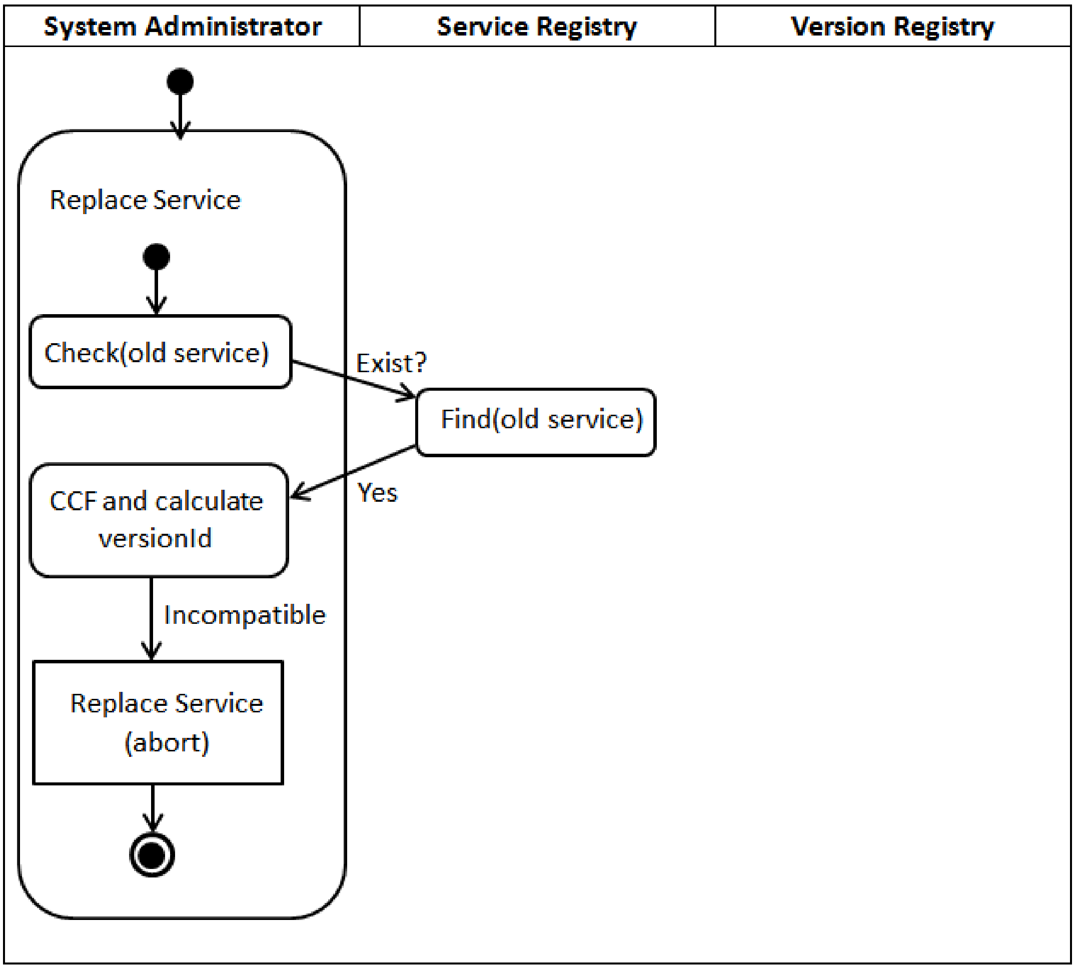


Figure 3.8: Activity diagram of replace service use case (incompatible)

Figure 3.8 describes an incompatible case of replacement operation. After finding the old service version, a CCF assessment is performed. In the case of incompatibility, the system will inform that "This service is an incompatible version, please try *unregisterService* and *registerNewService*, or *deployParallel* operations". This operation will then be aborted.

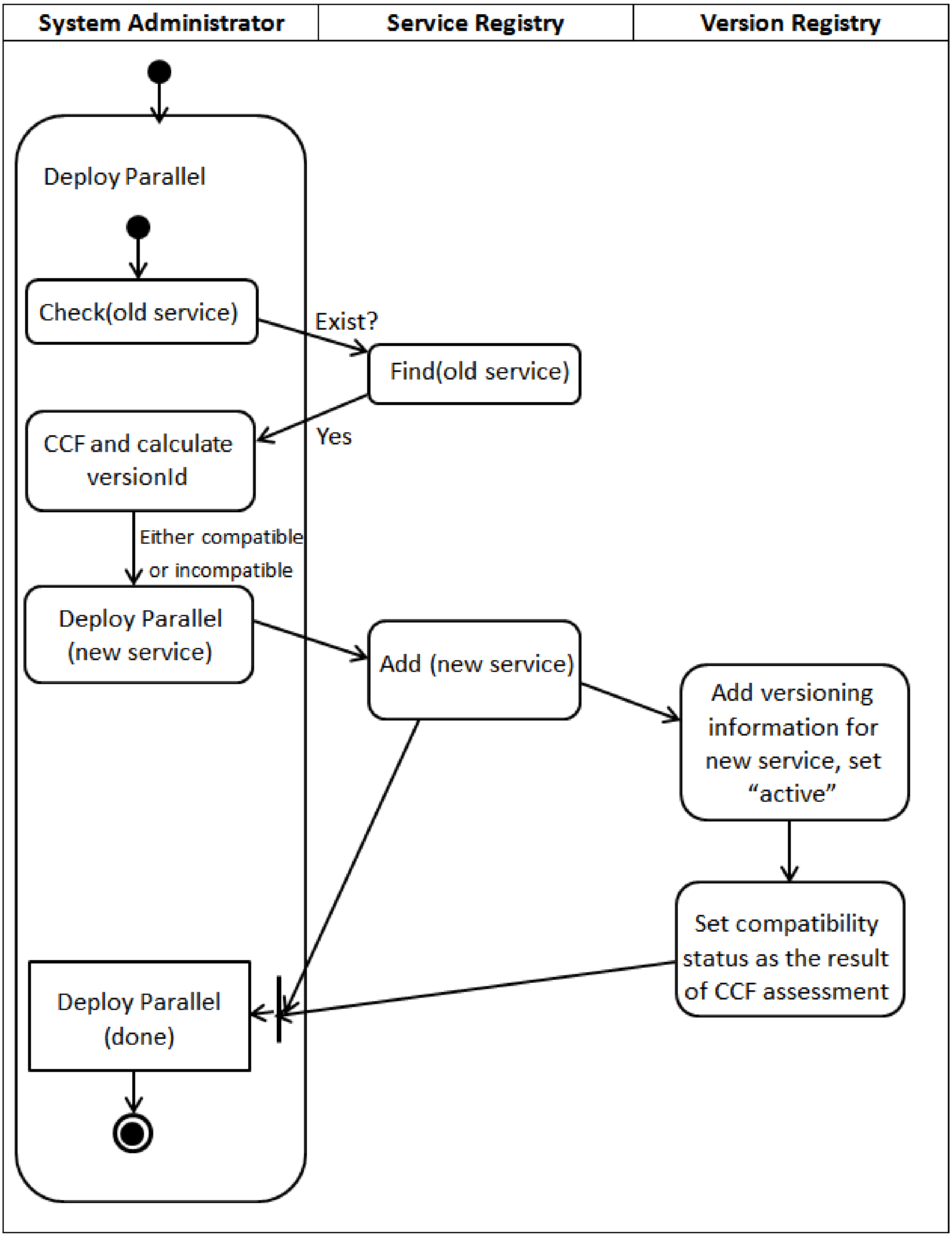

System Administrator
Service Registry
Version Registry
Deploy Parallel
Check(old service)
Exist?
Find(old service)
Yes
CCF and calculate versionId
Either compatible or incompatible
Deploy Parallel (new service)
Add (new service)
Add versioning information for new service, set "active"
Set compatibility status as the result of CCF assessment
Deploy Parallel (done)

Figure 3.9: Activity diagram of deploy service in parallel use case

Figure 3.9 shows the last use case for deploying a new service in parallel with an existing service. The system administrator registers directly the new service version into the Service Registry. This operation is similar to *registerNewService*. In addition, it requires a CCF assessment and an existence check for old service. If the related old service could not be found in the Service Registry, the system administrator will be informed about that "This old service version does not exist, please try *registerNewService* operation!" and all transactions will be then aborted. If such old service could be found, the process works continuously. The result of the CCF assessment is used for calculating the service version identifier and setting compatibility status in the Version Registry. The operation is successfully completed after the registries have stored the new service.

3.3.4 The formal expression of service descriptions for implementing CCF algorithm

In Section 2.8.2 the CCF assessment for checking the compatibility of services is introduced. For implementing it, we will explain how to express a service description based on ASD meta-model. This service description is only related to the structural layer of the ASD meta-model.

A structural ASD includes elements and their relationships. The relationships indicate the structural dependences among the elements. The formal definition for elements and their relationships has been proposed by [ABP12].

- An element e is a tuple e := (name: string, ($att_{i,i\geq1:}$: attribute)*, ($pr_{j,j\geq1:}$: property)*).
- A relationship r(e_s, e_t) between element e_s (the source element) and e_t (the target element) is a tuple := ($name_s$: string, $name_t$: string, rel: relation, mul: multiplicity).
- Where
  - name, $name_s$, $name_t$ are the unique element identifiers of elements e, e_s, e_t, respectively (of type string).
  - (att_i, i = 1,. . . ,m)* a set of zero or more generic types of attributes (int, char, string, etc.).
  - (pr_j, i = 1,. . . ,n)* a set of zero or more property value, that is, attributes with predefined value ranges and characteristics.
  - rel is the type of relation between the elements (a, c, s – aggregation, composition, or association with the semantic defined in [ABP08]).
  - mul is the multiplicity of the relationship, defined as mul := [min_{crd}, max_{crd}] where min_{crd}, max_{crd} ∈ IN (the set of natural numbers) is the minimum and maximum, respectively, multiplicities allow for each member of relationship.

According to this definition, we can further define the structural subtyping as follows [ABP12].

- For e = (name, att$_1$,...,att$_k$, pr$_1$,...,pr$_l$) and e$'$ = (name$'$, att$'_1$,...,att$'_m$, pr$'_1$,...,pr$'_n$), we define the subtype relation between e and e$'$ as

$$e \leq e' \Leftrightarrow name \equiv name' \bigwedge$$

$$k > m, att_i \leq att'_i, 1 \leq i \leq m \bigwedge$$

$$l > n, pr_j \leq pr'_j, 1 \leq j \leq n.$$

  This means that the name identifiers of elements must be identical, and the number of attributes and properties contained by e$'$ is respectively smaller than that contained by e, but the one of e$'$ is a supertype of the respective attributes and properties of e. By definition it holds that $(e = \emptyset) \leq e'$.

- For r(e$_s$, e$_t$) = (name$_s$, name$_t$, rel, mul) and r(e$'_s$, e$'_t$) = (name$'_s$, name$'_t$, rel$'$, mul$'$), we define the subtype between r and r$'$ as

$$r(e_s, e_t) \leq r(e'_s, e'_t) \Leftrightarrow e_s \leq e'_s \bigwedge e_t \leq e'_t \bigwedge rel = rel' \bigwedge mul \subseteq mul'.$$

  This denotes that the elements e$'_s$ and e$'_t$ which participates in the new relationship is respectively a supertype of e$_s$ and e$_t$ and the multiplicity domain of the relation is a superset of the respective one in the old relationship. As an assumption, $\emptyset \leq r(e_s, e_t)$, iff mul=[0, N), N $\geq$ 1.

These definitions enable us to express service descriptions formally by abstracting its elements and the relationships of them. Therefore, the CCF algorithm in Algorithm 2.1 is able to be implemented.

3.3.5 Dynamic routing of messages

Because multiple service versions might exist at the same time or an existing service version is probably replaced by another service version, a request message from clients or a response from services could not be successfully routed by ESBs. Therefore, the capabilities of the multi-tenant aware ESB have to be improved to ensure routing correctness.

An ESB has two main components namely Binding Components and Service Engines. We will explain the extension of the ESB from these two aspects.

3.3.5.1 Binding Components of the multi-tenant ESB

In the Service Registry the *versionedServiceName* distinguishes the multiple versions of a service from each other. It could be used as *serviceLocationPart* in URI to unambiguously locate a service version. In the case of replacement, the URI of the decommissioned service will be unchangingly assigned to the new active service. Therefore, the Binding Components which point to various service versions do not need to be changed.

3.3.5.2 Service Engines of the multi-tenant ESB

A Service Engine should contain a message interceptor, which is responsible for listening to the communication channel, intercepting each request from service consumers and each response from service providers, determining the real targeted service version of each intercepted message, and if necessary, invoking the transformer to convert incoming message to corresponding outgoing message, and vice versa (see Figure 3.10).

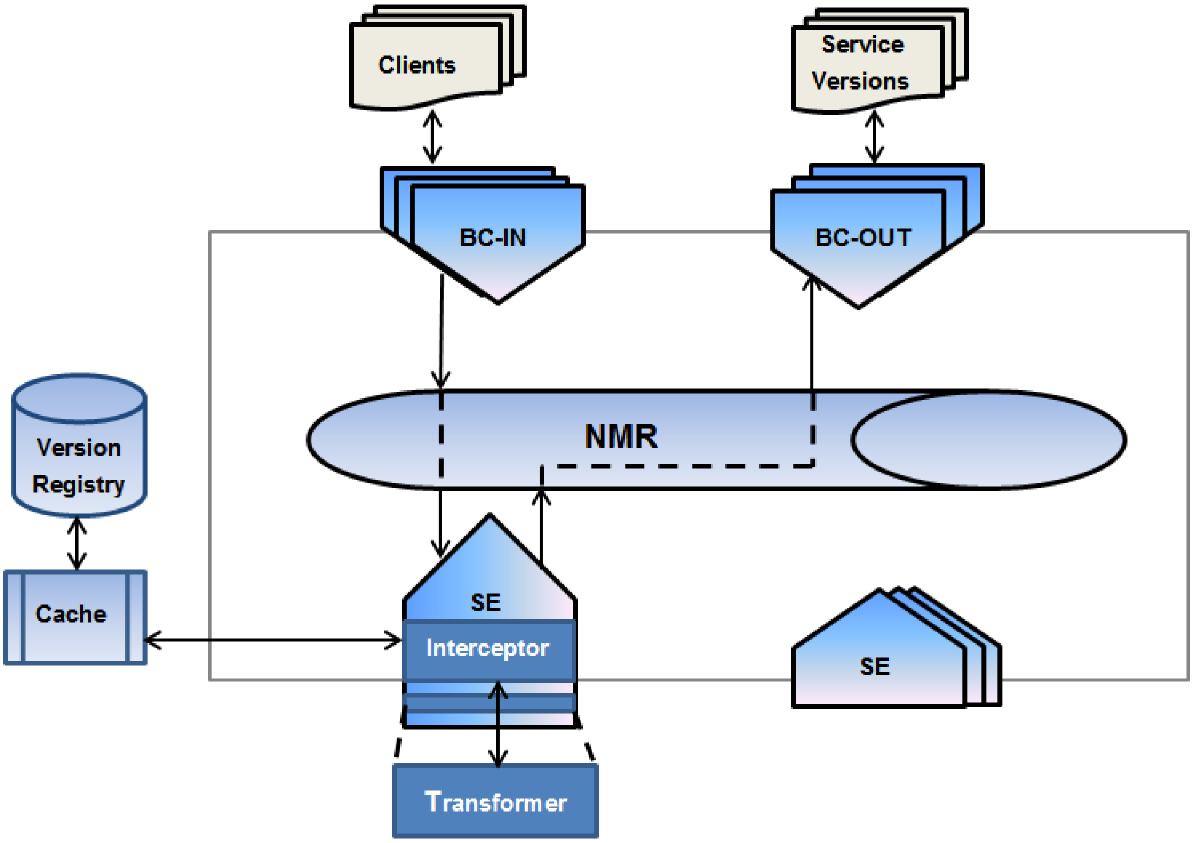


Figure 3.10: Extending SE for dynamic routing

The compatibility status is here important information. It could be retrieved by the interceptor to determine the targeted service of a message. The business logic of a message interceptor is described in Listing 3.1. We assume a message is targeted at a service named exampleService#m.n where m is the major release number and n is the minor release number.

The Listing 3.1 indicates four main cases for processing messages. If an identical service is found in the Service Registry, the message intercepted by the message interceptor could be directly routed to this service endpoint. If a compatible service version is returned, the SE is able to deliver this converted message through the NMR to its corresponding target, after the conversion has been executed by the transformer. In the case of incompatible service, the system will send a message to inform that the client should be updated in time. When the targeted service does not exist in the Version Registry (anymore), the client will be noticed.

Listing 3.1 Business logic of a message interceptor

```
A message is intercepted, its target service: "exampleService#m.n"

IF exampleService#m.n exist in the Version Registry THEN{
       IF serviceStatus = "active" THEN {
          SE delivers the message to the endpoint;
        }ELSE {
              IF at least one compatible service version exist THEN {
                find the latest service version by comparing "startTime";
                invoke transformer;
                SE delivers the converted message to the target service;
               } ELSEIF at least one incompatible service version exist THEN {
                 notice "update client";
               } ELSE { notice "this service does not exist anymore";
               }END IF
       }END IF
 } ELSE {notice "this service does not exist";
 } END IF
```

A transformer is required to convert the request messages of service consumers to the corresponding messages supported by compatible service versions and vice versa. It can be embedded into a Service Engine or designed as a functionality of the service version control management system. If a transformer is invoked by a message interceptor, it adapts an incoming message as below. We assume that a service named *servicePre* is replaced by a compatible version named *serviceCurr* and a request message from service consumer is targeted at the service *servicePre*.

Listing 3.2 Business logic of a transformer

```
get the incoming message from SE;
get the WSDL file of servicePre from Version Registry;
get the WSDL file of serviceCurr from Version Registry;
IF incoming message = request message from service consumer THEN {
       FOR each input-type element e in request message{
              find the corresponding input-type element e’ in WSDL file of serviceCurr;
              adapt e to e’,
       }END FOR
       return the resulted message to SE;
}END IF
IF incoming message = response message from service provider THEN{
       FOR each out-put element e’ in response message{
              find the corresponding out-put element e in WSDL file of servicePre;
              adapt e’ to e,
       } END FOR
       return the resulted message to SE;
}END IF
```

3.3.5.3 Message processing logic

A cache is located between the Version Registry and the message interceptor for keeping the retrieved results for a while, in order to reduce the retrieval time and frequency. It stores the queried version-related information from the Version Registry, such as the compatibility status and its associated two service versions.

A request message from the consumer site is normalized by the BC-IN and wired to a corresponding SE. The message interceptor of the SE intercepts the normalized message and retrieves the versioning information from the Version Registry or from the cache (if the latest results of retrievals still exist). Based on the retrieved information, the message interceptor can decide whether it invokes the transformer. The SE forwards then the formalized message through the NMR to the BC-OUT.

Similar to the request message, a response message from the provider side is normalized by the BC-OUT and wired to a corresponding SE. According to the service versioning information from the cache, the message interceptor decides whether the message payload needs to be transformed before forwarding. The normalized message is finally routed to the BC-IN where the request message came from.

In the previous section the four cases have been determined based on the Listing 3.1, namely "identical", "compatible", "incompatible" and "not exist (anymore)". For a better understanding of these four cases, we combine sequence diagrams about message flows to illustrate how an incoming message targeted at a particular service endpoint is being processed. It also explains how the message interceptor processes messages.

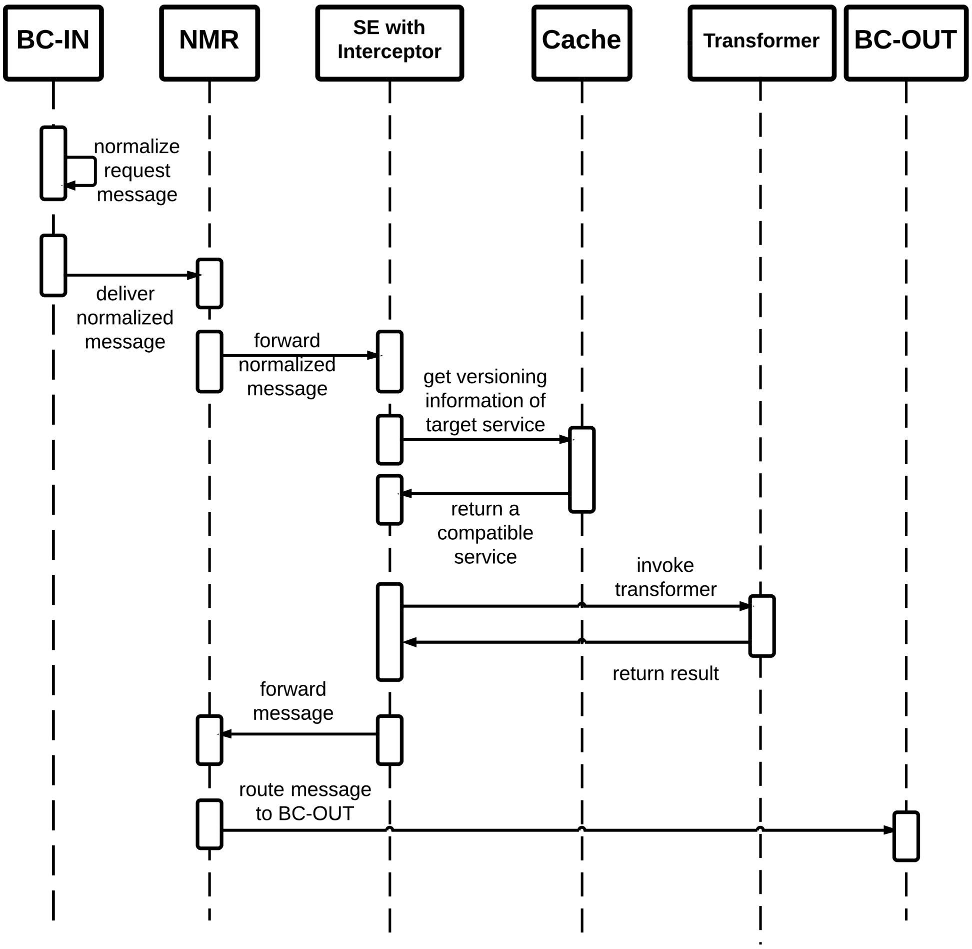

BC-IN
NMR
SE with Interceptor
Cache
Transformer
BC-OUT
normalize request message
deliver normalized message
forward normalized message
get versioning information of target service
return a compatible service
invoke transformer
return result
forward message
route message to BC-OUT

Figure 3.11: Sequence Diagram in the case of "compatible"

The Figure 3.11 shows a sequence diagram for explaining the behaviors of the system components, in case that a compatible service is considered as the target service. An incoming message from BC-IN is normalized and wired to the corresponding SE of the BC-IN. The message interceptor of the SE obtains a compatible service and uses it as the new target service of the normalized massage. After being converted into the compatible format, the message is forwarded to the NMR by SE and then routed to the corresponding BC-OUT by NMR.

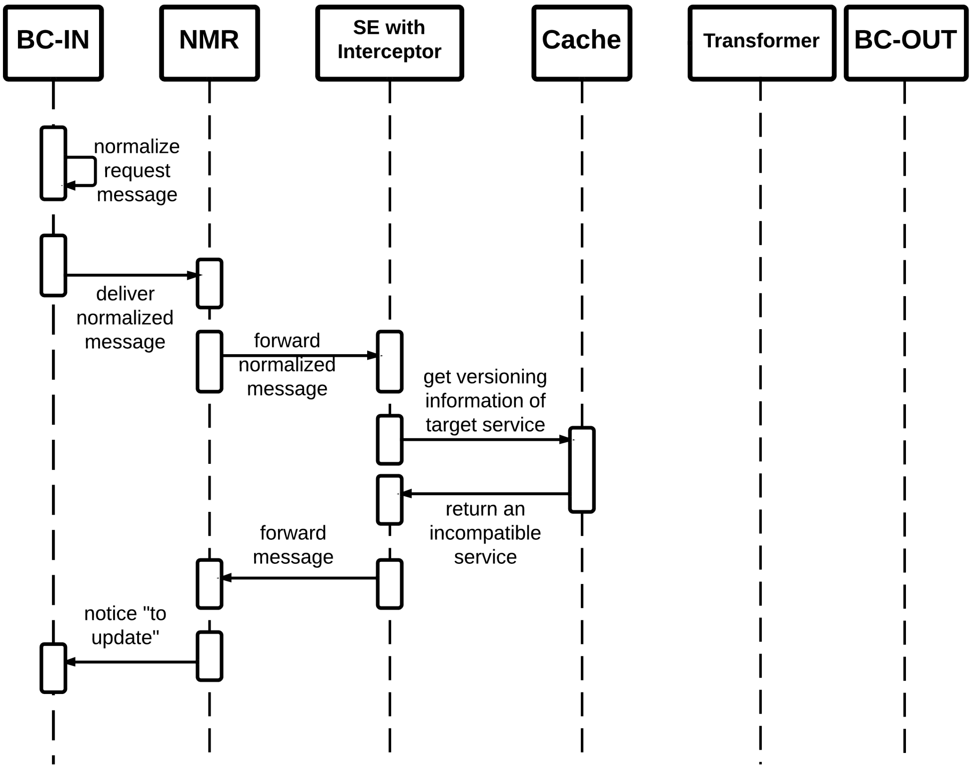

BC-IN
NMR
SE with Interceptor
Cache
Transformer
BC-OUT
normalize request message
deliver normalized message
forward normalized message
get versioning information of target service
return an incompatible service
forward message
notice "to update"

Figure 3.12: Sequence Diagram in the case of "incompatible"

The Figure 3.12 is related to an incompatible case. After the normalization, the message is intercepted for retrieving its target service. If the resulted service is an incompatible version, a notification "update the client" will be returned to the BC-IN.

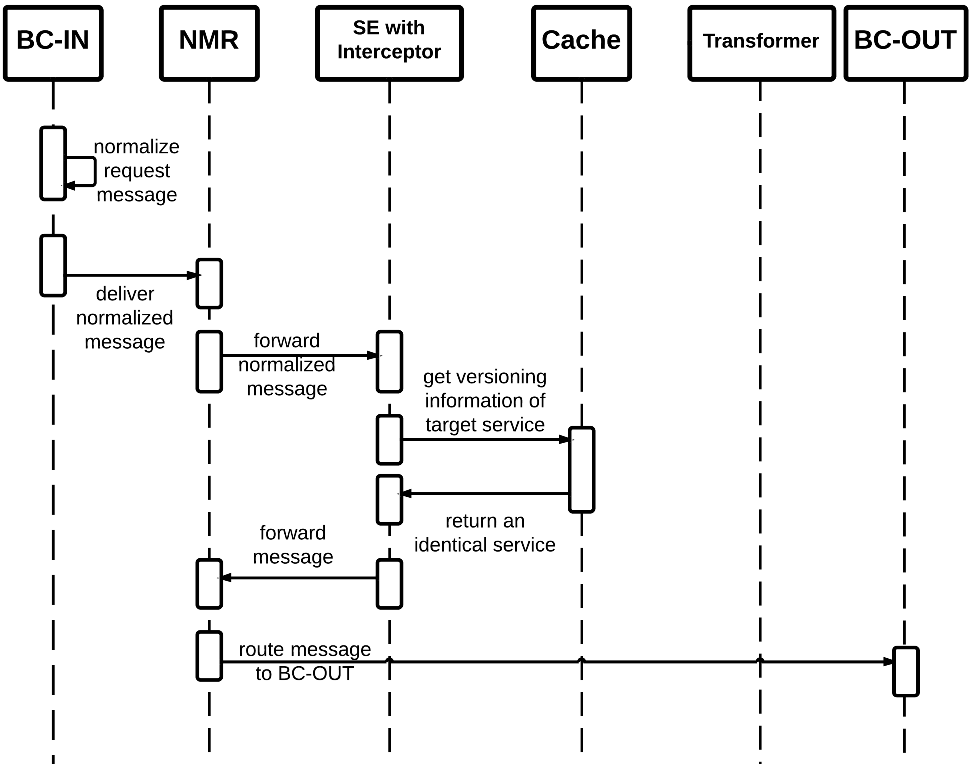

BC-IN
NMR
SE with Interceptor
Cache
Transformer
BC-OUT
normalize request message
deliver normalized message
forward normalized message
get versioning information of target service
return an identical service
forward message
route message to BC-OUT

Figure 3.13: Sequence Diagram in the case of "identical"

The Figure 3.13 explains the behaviors of the system components after an identical service is being retrieved by the message interceptor. The SE which supports the routing functionality can route the normalized incoming message directly to the corresponding BC-OUT.

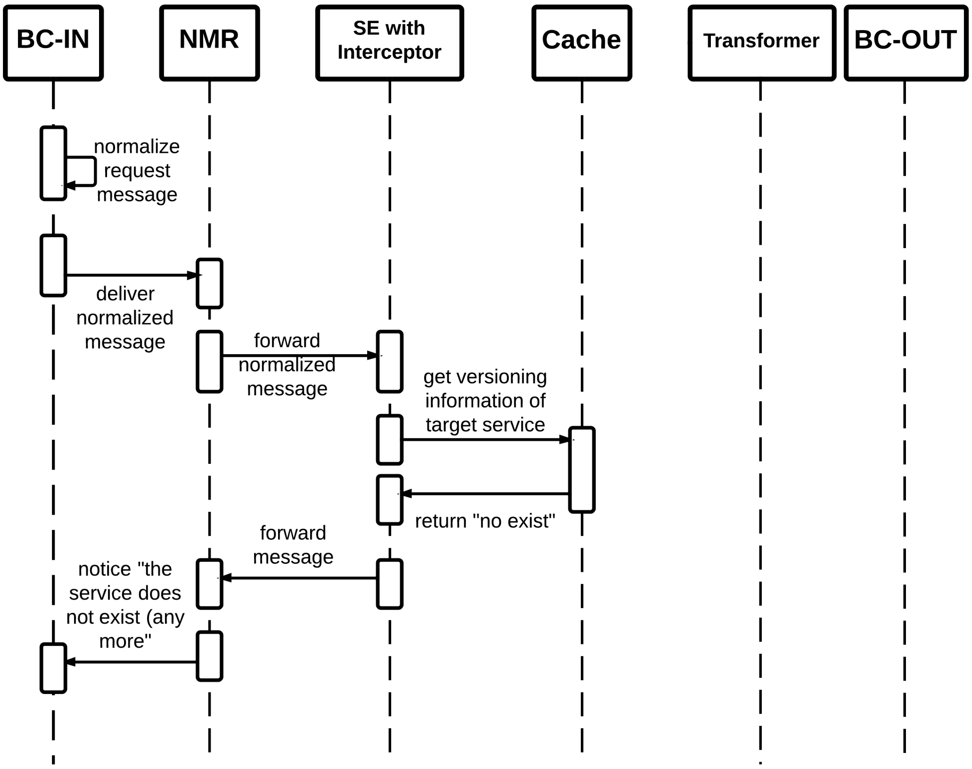

BC-IN
NMR
SE with Interceptor
Cache
Transformer
BC-OUT
normalize request message
deliver normalized message
forward normalized message
get versioning information of target service
return "no exist"
forward message
notice "the service does not exist (any more"

Figure 3.14: Sequence Diagram in the case of "not exist (anymore)"

The last case is that the message inceptor cannot get any versioning information about the normalized incoming message. That means the target service of the message does not exist (anymore). Therefore the system returns a notification to the BC-IN.

3.4 Summary

In this chapter the requirements of the service version control management system is identified. In order to better understand the functionalities of this system, three use cases are introduced and concretely explained by activity diagrams. In Section 3.3 the JBIMulti2 management system is extended for managing service version purpose. Additionally, the CCF algorithm could be performed by the formal expression of service descriptions. In the end, the Service Engine which supports the routing compatibility is improved by adding a message interceptor which is combined with a transformer to implement the dynamic routing of messages. The behaviors of the system components are explained by sequence diagrams for various cases.

4 Implementation

This chapter describes firstly the foundational technologies which are related to the extension of the JBIMulti2 system. Then we explain how to implement the Version Registry and the CCF assessment function, and how the functionalities of the JBIMulti2 are extended for service evolution management.

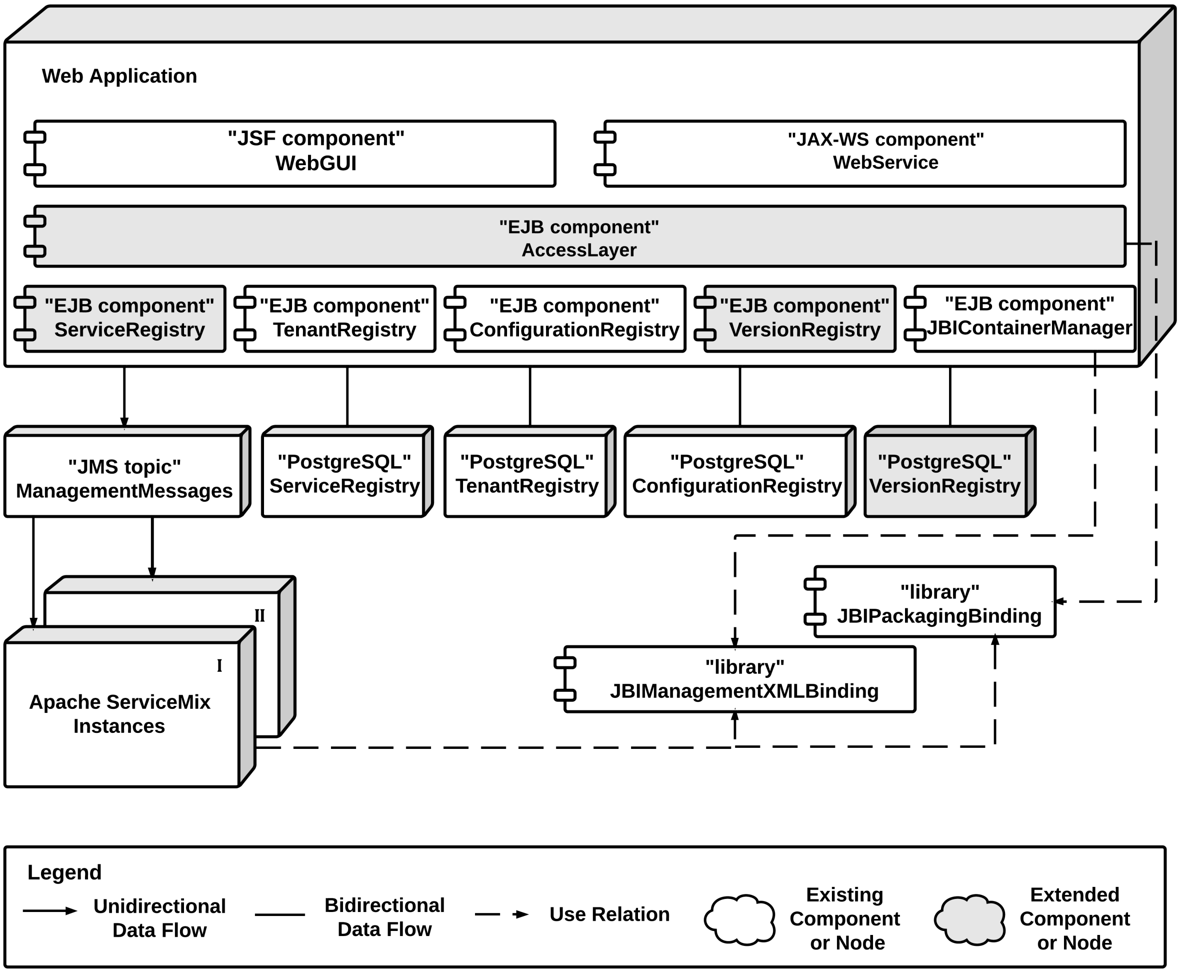


Figure 4.1: Overview of the extended JBIMulti2 architecture

Figure 4.1 is an overview of the service version control management system which reveals the technological architecture. It comprises system components and associated technologies.

4.1 Fundamental Technologies

We introduce in this section the technologies related to the extension of the JBIMulti2 system for service evolution management.

Java Platform, Enterprise Edition v.5 (Java EE 5) offers a powerful set of APIs to reduce development time and application complexity, as well as improving application performance. It simplifies the development of distributed, transactional, and portable applications [FJB10]. The following APIs are necessary for this implementation.

- Java API for XML-based Web Services (JAX-WS)
  JAX-WS is used to develop Web Services and clients that use XML-based protocol such as SOAP. The complexity of SOAP is here irrelevant, because it could be hidden by JAX-WS from the application developers. In addition, JAX-WS maps Java interfaces to WSDL and vice versa.
- Java Architecture for XML Binding (JAXB)
  A fast and appropriate method is provided by JAXB to transform XML documents to or from Java objects.
- Java Message Service (JMS)
  The JMS specifies a set of interfaces and associated semantics for Java developers to integrate messaging products. It enables communications to be loosely coupled, asynchronous, and reliable.
- Enterprise JavaBeans (EJB)
  An EJB is a server side component. It encapsulates the business logic which performs the functionalities of an application. The bean developer could focus on solving business problems because the EJB container is responsible for system-level services. Thus, the development of large, distributed applications has become easier. Additionally, the client developer could design thin clients running on small devices, since the business logic of the application is contained by the EJB on the server side. The EJB provides also scalability of applications, data integrity of transactions, and variety of clients.
- JavaService Faces (JSF)
  This is a user interface component framework on server side for web applications. It provides architecture for managing component state, processing component data, validating user input, and handling events.

PostgreSQL is an open-source object-relational database management system [Pos]. It provides ACID compliance. Moreover, it is cross-platform and can almost run on major operating systems. As a database server, we use it to manage the data of Service Registry, Version Registry, Tenant Registry, and Configuration Registry.

Apache ServiceMixis an open-source distributed ESB. It integrates the functionalities of a SOA model and the modularity of OGSi framework to decouple the applications and reduce their dependencies [Fouc]. Its components such as Apache Karaf, Apache ActiveMQ and Apache Camel have been described in Section 2.6.

Apache Maven is a software project management tool. It provides programmers with a complete build lifecycle framework. Various java projects could be built during their build lifecycle by resolving the Project Object Model (POM) file which contains the tasks and required plugins of projects. Furthermore, Apache Maven could specify the dependencies between modules stored in local or remote repository to manage a large-scale project [Foub].In this thesis, the module related to the JBIMulti2 web application is extended to support service evolution management. All java source code is compiled with the Java Development Kit (JDK) 6.

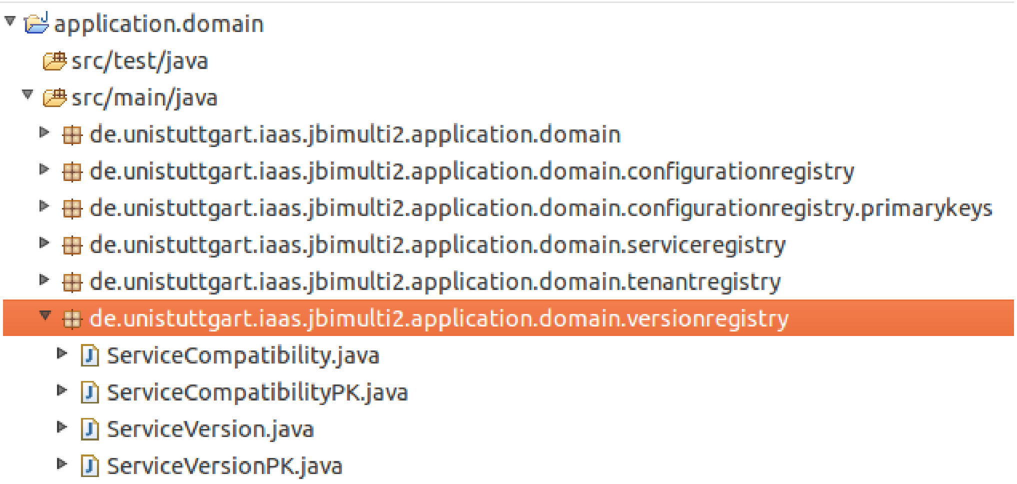
4.2 Implementation of Version Registry

The Version Registry described in Section 3.3.2.1 is implemented by PostgreSQL. Listing 4.1 shows its Data Definition Language (DDL). The source database consists of two tables. In the table of *service_compatibility*, *service_name_pre* and *service_version_id_pre* identify a previous service version which for example should be replaced, while *service_name_curr* and *service_version_id_curr* identify the new registered service version. They associate *compatibility_status* to record the compatibility of two services.

Listing 4.1 DDL of Version Registry

```
CREATE DATABASE versionregistry;
CREATE TABLE service_version (
       service_version_id                  varchar(255) NOT NULL,
       service_name                        varchar(255) NOT NULL,
       service_status                      varchar(20),
       start_time                          date,
       end_time                            date,
       wsdlFile                            text NOT NULL,
       PRIMARY KEY (service_version_id, service_name)
);
CREATE TABLE service_compatibility (
       service_name_prev           varchar(255) NOT NULL,
       service_version_id_prev     varchar(255) NOT NULL,
       service_name_curr           varchar(255) NOT NULL,
       service_version_id_curr     varchar(255) NOT NULL ,
       compatibility_status        varchar(20) NOT NULL,
       PRIMARY KEY (service_name_prev, service_version_id_prev, service_name_curr,
           service_version_id_curr),
       FOREIGN KEY (service_name_prev, service_version_id_prev) REFERENCES service_version
           (service_name, service_version_id),
       FOREIGN KEY (service_name_curr, service_version_id_curr) REFERENCES service_version
           (service_name, service_version_id)
);
```

The Java Persistence API (JPA) provides an object-relational mapping facility to manage relational data in Java applications. The entities and entity relationships can be mapped into the relational data in the underlying database by using object-relational mapping annotations (*javax.persistence.**). An entity class e.g. *ServiceVersion* (*service_version* entity) must be annotated with @*Entity* annotation. @*EmbeddedId* annotation denotes the composite primary key property or field of an Entity, for example, *compositePrimaryKey* of *ServiceVersion* entity consists of *service_version_id* and *service_name*. The EntityManager API manages entity beans by creating or removing persistence entity instances, finding entities with the primary key, and allowing queries to be on entities.



```
▼ application.domain
    src/test/java
  ▼ src/main/java
    ▶ de.unistuttgart.iaas.jbimulti2.application.domain
    ▶ de.unistuttgart.iaas.jbimulti2.application.domain.configurationregistry
    ▶ de.unistuttgart.iaas.jbimulti2.application.domain.configurationregistry.primarykeys
    ▶ de.unistuttgart.iaas.jbimulti2.application.domain.serviceregistry
    ▶ de.unistuttgart.iaas.jbimulti2.application.domain.tenantregistry
    ▼ de.unistuttgart.iaas.jbimulti2.application.domain.versionregistry
      ▶ ServiceCompatibility.java
      ▶ ServiceCompatibilityPK.java
      ▶ ServiceVersion.java
      ▶ ServiceVersionPK.java
```

Figure 4.2: Relational data in Java to be mapped

Figure 4.2 lists four java files under the package *domain.versionregistry*. The *ServiceVeresionPK* and *ServiceCompatibilityPK* define the composite primary key for *ServiceVersion* and *ServiceCompatibility* which are mapped to *service_version* and *service_compatibility* respectively (see Listing 4.1).

Listing 4.2 Persistence unit of *versionRegistry*

```
<persistence-unit name="versionRegistry" transaction-type="JTA">
<provider>org.hibernate.ejb.HibernatePersistence</provider>
<jta-data-source>jbimulti2/versionRegistry</jta-data-source>
<class>de.unistuttgart.iaas.jbimulti2.application.domain.
                                    versionregistry.ServiceCompatibility</class>
<class>de.unistuttgart.iaas.jbimulti2.application.domain.
                                    versionregistry.ServiceCompatibilityPK</class>
<class>de.unistuttgart.iaas.jbimulti2.application.domain.
                                    versionregistry.ServiceVersion</class>
<class>de.unistuttgart.iaas.jbimulti2.application.domain.
                                    versionregistry.ServiceVersionPK</class>
......
</persistence-unit>
```

A JPA persistence unit is a logical grouping of user defined persistable classes with related setting. A *persistence.xml* in the *META-INF* directory in the classpath defines persistence units. Listing 4.2 shows how to define *versionRegistry* as a JPA persistence unit.

4.3 Compatibility Check

A WSDL file contained in the Service Registry is in Character Large Object (CLOB) format. We need firstly a *printer* to transform the WSDL string into XML-based WSDL document.

Listing 4.3 Implementation of CCF function

```
input: reqElements1, reqRelationships1, proElements1, proRelationships1 for Service1
       reqElements2, reqRelationships2, proElements2, proRelationships2 for Service2

boolean compatibility = true;
// CCF Algorithm line 1-5
// check the subtyping of elements
FOR each element reqEle1 of reqElements1 {
    IF ((name of reqEle1)== (name of reqEle2) &&
        (type of reqEle1) is subtype (type of reqEle2)) {
         compatibility = true;
     } ELSE {compatibility = false; break;}
}
//check the subtyping of relationships
FOR each relationship reqRel1 of reqRelationships1 {
    IF ((source name of reqRel1) == (source name of reqRel2) &&
       (target name of reqRel1) == (target name of reqRel2) &&
       (multiplicity of reqRel1) is subtype of (multiplicity of reqRel2)) {
         compatibility = true;
     } ELSE {compatibility = false; break;}
}

// CCF Algorithm line 6-10
// check the subtyping of elements
FOR each element proEle2 of proElements2 {
    IF ((name of proEle1) == (name of proEle2) &&
       (type of proEle2) is subtype (type of proEle1)) {
         compatibility = true;
     } ELSE {compatibility = false; break;}
}
//check the subtyping of relationships
FOR each relationship proRel2 of proRelationships2 {
    IF ((source name of proRel1) == (source name of proRel2) &&
       (target name of proRel1) == (target name of proRel2) &&
       (multiplicity of proRel2) is subtype of (multiplicity of proRel1)) {
         compatibility = true;
     } ELSE {compatibility = false; break;}
}

return compatibility;
```

A WSDL file is an XML-based document. According to the formal expression of service descriptions proposed in Section 3.3.4, the functions of *ASD* class could analyze XML element-tags in order to convert a XML-based WSDL document into a formal structural ASD expression. In other words, the elements and their relationships can be extracted by *getAllElements* and *getAllRelationships* respectively. Furthermore, all extracted elements could be classified as input-type or output-type. The input-type elements, output-type elements, and their corresponding relationships enable the CCF assessment to be implemented.

We assume that *reqElements* and *reqRelationships* save separately all input-type elements and their relationships of a service description, while *proElements* and *proRelationships* store all output-type elements and their relationships respectively. For subtype ($\leq$) of data type, we assume int $\leq$ double $\leq$ string $\leq$ document. For subtype ($\leq$) of multiplicity, we assume [1, 1] $\leq$ [0, 1], where [1, 1] denotes a mandatory appearance of an element and [0, 1] denotes an optional appearance of an element.

4.4 Implementation of Service Version Control Management

Two EJB components namely *VersionRegistry* and *ServiceRegistry*(see Figure 4.1) are separately realized to encapsulate the business logic to access the Version Registry and the Service Registry.

The *VersionRegistry* invokes the EntityManager to add a new service version into the Version Registry, to query version-related information with the service name and version identifier, or to add or delete service compatibility into or from the Service Registry respectively, etc.

The *ServiceRegistryis* extended by adding two operations to the Service Registry. It is able to remove a previous service version and add the new service version into the Service Registry. Moreover, it could deploy a new service version in parallel with an existing service version. Special attention should be paid to the composite key of a service entity, which consists of tenant identifier, user identifier and service name. Although the tenant identifier and user identifier are optional, the *null* value could lead to errors in database or in operations. Therefore, we set both as "" value.

The JBIMulti2 system should provide three services to manage the service evolution. This has been explained in three use cases in Chapter 3. For the same reason, the *SystemAdminFaçadeBean* is extended by adding *registerNewService*, *replaceService* and *deployParallel* functionalities. Their business methods are described in Listing 4.4, Listing 4.5 and Listing 4.6, respectively.

The business methods of façade enterprise beans have to be annotated with @*PermissionType* to inform the method interceptor which permission a caller must have. A *parser* is designed to read the service name from a WSDL file. Assertion functions are introduced to check whether the related service has already existed. Based on the version identifier of the previous service version and the result of CCF assessment, the version identifier of the new service version can be calculated with the *versionIdCalculate* function (see Listing 4.7).

Listing 4.4 The business method of *registerNewService* in *SystemAdminFaçadeBean*

```
@PermissionType(type = PermissionTypeEnum.SYSTEM_ADMINISTRATOR)
public ServiceEntry registerNewService (String wsdlFile)
       throws AuthorizationException, ExecutionException{
       ...
       String serviceName = parser.getServiceName(wsdlName);
       String versionedServiceName = serviceName + "#" + "1.0";
       //check if an active service with serviceName does exist in the Version Registry
       assertionManager.assertServiceNameExists(serviceName);
       //if such service does not exist in the Version Registry
       serviceRegistry.registerService("", "", versionedServiceName, wsdlFile);
       versionRegsitry.addServiceVersion("1.0", "serviceName", "active", startTime);
       ...
}
```

Listing 4.5 The business method of *replaceService* in *SystemAdminFaçadeBean*

```
@PermissionType(type = PermissionTypeEnum.SYSTEM_ADMINISTRATOR)
public ServiceEntry replaceService (String versionedServiceNamePre,
       String wsdlFileCurr) throws AuthorizationException, ExecutionException{
       ...
       //check if this service to be replaced does exist in the Service Registry
       assertioManager.assertServiceNotExists(versionServiceNamePre);
       //if this service does exist in the Service Registry
       get serviceName and versionIdPre from versionedServiceNamePre;
       String wsdlFilePre = serviceRegistry.getService("", "",
           versionedServiceName).getWsdlFile();
       //CCF assessment
       String compatibilityStatus = CCF(wsdlFilePre, wsdlFileCurr);
       if (compatibilityStatus == "compatible"){
           String veresionIdCurr = versionIdCalculate(versionIdPre, compatibilityStatus);
           String versionedServiceNameCurr = serviceName + "#" + versionIdCurr;
           versionRegistry.replaceService(versionedServiceNamePre, versionedServiceNameCurr,
               wsdlFileCurr);
           set serviceStatus of servicePre as "decommissioned" in the Version Registry;
           set entTime of servicePre in the Version Registry;
           versionRegistry.addServiceVersion(versionIdCurr, serviceName, "active", startTime);
           add compatibilityStatus with associated composite key into service_compatibility
               table;
        }
       ...
}
```

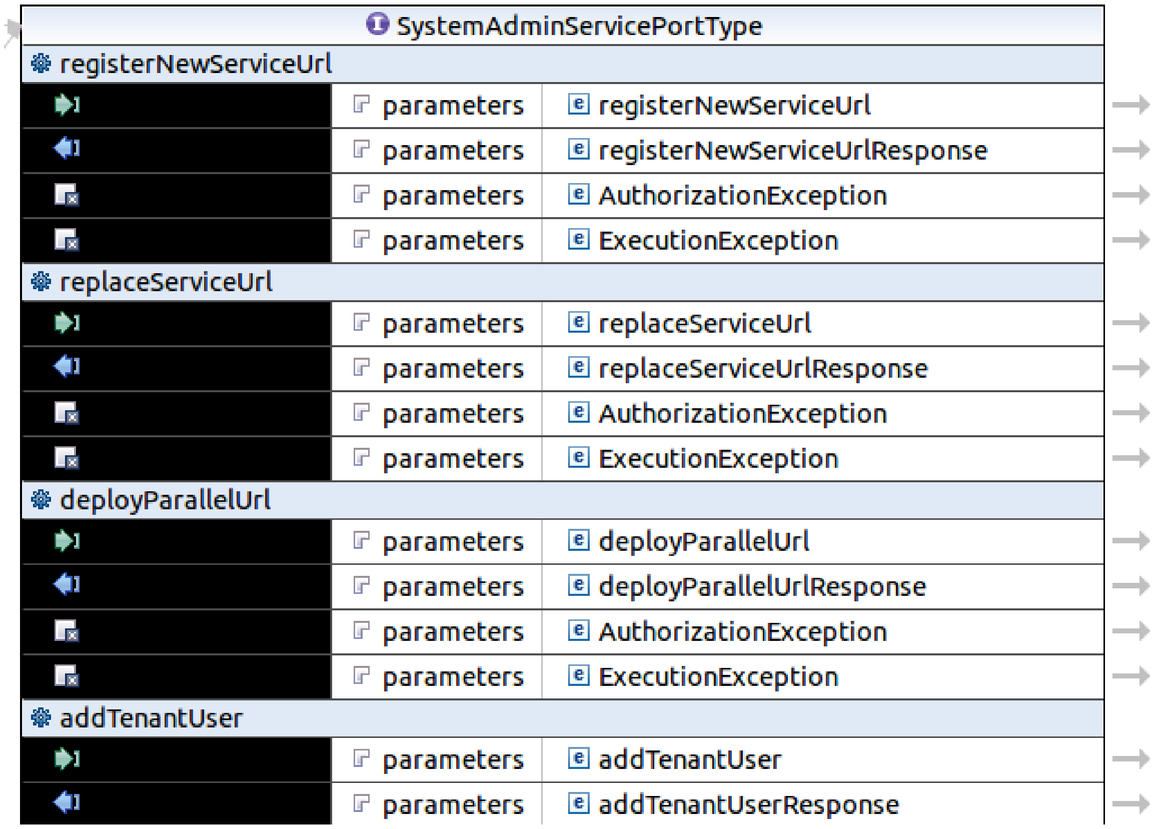
Listing 4.6 The business method of *deployParallel* in *SystemAdminFaçadeBean*

```
@PermissionType(type = PermissionTypeEnum.SYSTEM_ADMINISTRATOR)
public ServiceEntry deployParallel (String versionedServiceNamePre,
       String wsdlFileCurr) throws AuthorizationException, ExecutionException{
       ...
       //check if this related service does exist in the Service Registry
       assertioManager.assertServiceNotExists(versionServiceNamePre);
       //if this service does exist in the Service Registry
       get serviceName and versionIdPre from versionedServiceNamePre;
       String wsdlFilePre = serviceRegistry.getService("", "",
           versionedServiceName).getWsdlFile();
       //CCF assessment
       String compatibilityStatus = CCF(wsdlFilePre, wsdlFileCurr);
       String veresionIdCurr = versionIdCalculate(versionIdPre, compatibilityStatus);
       String versionedServiceNameCurr = serviceName + "#" + versionIdCurr;
       versionRegistry.deployParallel(versionedServiceNameCurr, wsdlFileCurr);
       versionRegistry.addServiceVersion(versionIdCurr, serviceName, "active", startTime);
       add compatibilityStatus with associated composite key into service_compatibility
           table;
       ...
}
```

Listing 4.7 The algorithm of *versionIdCalculate*

```
IF (compatibilityStatus == "compatible"){
   the Minor of versionIdPre is increased by 1;
   String versionIdCurr = Major + "." + Minor;
} ELSE {
   the Major of versionIdPre is increased by 1;
   String versionIdCurr = Major + "." + "0";
}
```

The package *de.unistuttgart.iaas.jbimulti2.application.webservice.wsdl* contains all generated Java classes and interfaces. *SystemAdminServicePortTypeBean* and *jbimulti2.wsdl* should be modified correspondingly. After running Maven plugin, the structure of the Web Service is changed (see Figure 4.3).



SystemAdminServicePortType

registerNewServiceUrl		
	parameters	registerNewServiceUrl
	parameters	registerNewServiceUrlResponse
	parameters	AuthorizationException
	parameters	ExecutionException
replaceServiceUrl		
	parameters	replaceServiceUrl
	parameters	replaceServiceUrlResponse
	parameters	AuthorizationException
	parameters	ExecutionException
deployParallelUrl		
	parameters	deployParallelUrl
	parameters	deployParallelUrlResponse
	parameters	AuthorizationException
	parameters	ExecutionException
addTenantUser		
	parameters	addTenantUser
	parameters	addTenantUserResponse

Figure 4.3: Service interface of System Admin

5 Validation

In this chapter we validate the implementation of our service version management system as discussed in Chapter 4. In first section, the environment for the validation should be introduced. In second section, the CCF assessment function is tested with several service examples. In the last section, we validate the three version control services of the JBIMulti2 system.

5.1 Validation Environment

We use the following tools to establish the validation environment (see Figure 5.1).

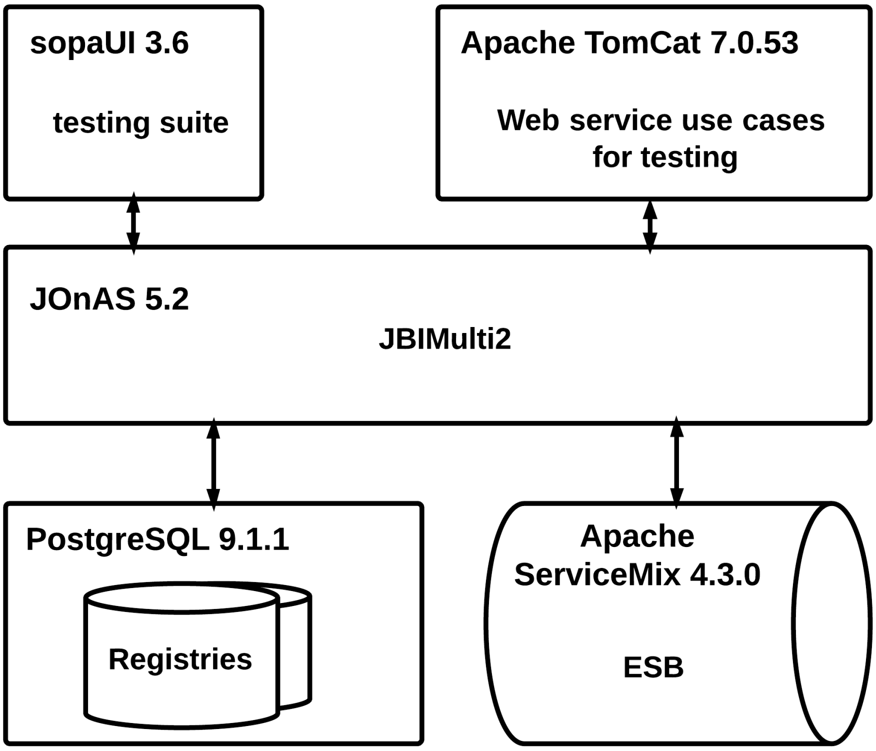


Figure 5.1: The validation environment

- PostgreSQL 9.1.1
  It is introduced in Section 4.1.All databases are created and managed by PostgreSQL 9.1.1.
- JOnAS 5.2 [Con]
  JOnAS is an open-source implementation of the Java EE application server. The JBIMulti2 EAR package is deployed by a JOnAS 5.2 instance. It makes a database connection to the PostgreSQL 9.1.1.
- SoapUI 3.6 [Sof]
  SoapUi is an open-source Web Service testing application. It provides a graphical interface to test Web Service functions of the extended JBIMulti2 system.
- Apache Tomcat 7.0.53 [Foud]
  Apache Tomcat is an open-source Web Server and servlet container. It hosts several service example deployments for testing CCF assessment function.
- Apache ServiceMix 4.3.0
  It integrates a messaging broker. Management messages from the JBIMulti2 management application are targeted at this message broker. The OSGi blueprint bundle *jbi.servicemix.jmsmanagement-1.0.jar* listens to these messages to perform JBI component installations and service assembly deployments. The management of JBI components and service assemblies is out of the scope of this work.

5.2 Validation of CCF assessment

In order to test the CCF assessment function, a Web Service called calculateService is developed and deployed by Apache Tomcat. An initial version calculateService1 has only two functionalities: addition and subtraction of two integer-type numbers. As an extended version, calculateService2 is added two operations: multiplication and division, but the input-types and output-types are still integer. Table 5.1-Table 5.5 describe the possible evolution paths of the calculateService. Based on the guideline-based approach discussed in Section 2.8.2, the version number of each service is deduced and it follows the service name in each table. It should be noticed that in Table 5.5 there are two deduced versions. Since all parameters are optional in default, the version number of calculateService should be 2.1. If the input parameter z is mandatory, the version number should be 3.0.

calculateService1 (1.0)	+	**-**
input x	int	int
input y	int	int
output	int	int

Table 5.1: calculateService1

calculateService2 (1.1)	+	-	×	÷
input x	int	int	int	int
input y	int	int	int	int
output	int	int	int	int

Table 5.2: calculateService2

calculateService3 (1.2)	+	-	×	÷
input x	double	double	double	double
input y	double	double	double	double
output	int	int	int	int

Table 5.3: calculateService3

calculateService4 (2.0)	+	-	×	÷
input x	double	double	double	double
input y	double	double	double	double
output	double	double	double	double

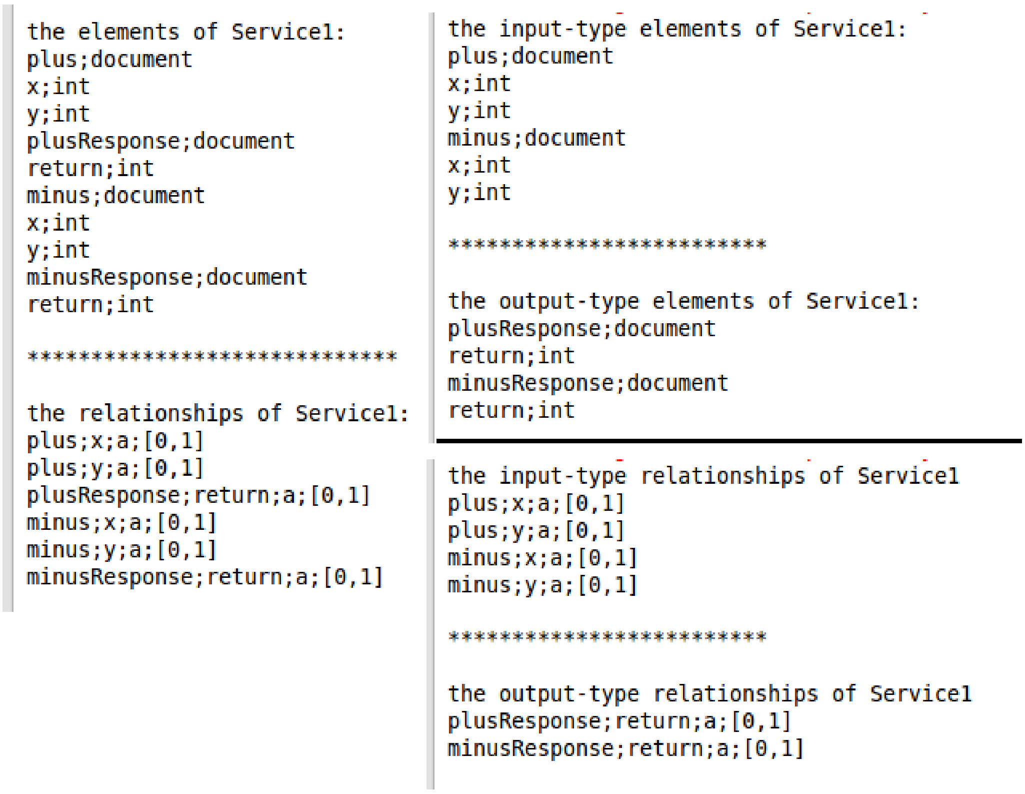
Table 5.4: calculateService4

calculateService5 (2.1/3.0)	+	-	×	÷
input x	double	double	double	double
input y	double	double	double	double
input z (optional/mandatory)	double	double	double	double
output	double	double	double	double

Table 5.5: calculateService5

The formal representation of a service description is the basis of implementing the CCF assessment. This is realized by analyzing the XML-based service descriptions of these calculation services. All records of a service description could be divided into input-type and output-type records. The ASD representation of calculateService1 is shown in Figure 5.2. On the left side, there are the elements and their relationships extracted from the WSDL file. They are divided into input- and output-types respectively, which are shown on the right side.

As a compatible example, Figure 5.3 shows the result of the CCF assessment which checks the compatibility of calculateService1 and calculateService2. In case of "compatible", the minor number will be increased by 1. That means the version of calculateService2 will then be set as 1.1.

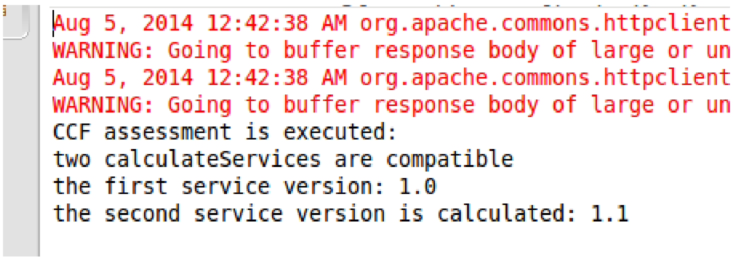


```
the elements of Service1:
plus;document
x;int
y;int
plusResponse;document
return;int
minus;document
x;int
y;int
minusResponse;document
return;int

*****************************

the relationships of Service1:
plus;x;a;[0,1]
plus;y;a;[0,1]
plusResponse;return;a;[0,1]
minus;x;a;[0,1]
minus;y;a;[0,1]
minusResponse;return;a;[0,1]
```

```
the input-type elements of Service1:
plus;document
x;int
y;int
minus;document
x;int
y;int

*************************

the output-type elements of Service1:
plusResponse;document
return;int
minusResponse;document
return;int
```

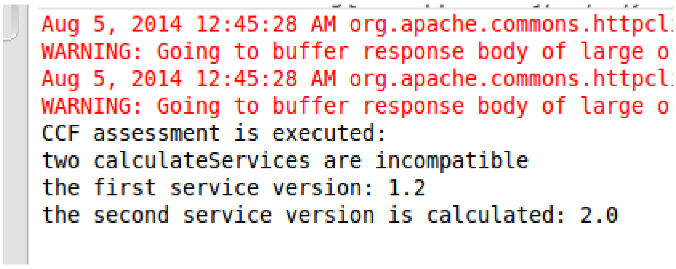
```
the input-type relationships of Service1
plus;x;a;[0,1]
plus;y;a;[0,1]
minus;x;a;[0,1]
minus;y;a;[0,1]

*************************

the output-type relationships of Service1
plusResponse;return;a;[0,1]
minusResponse;return;a;[0,1]
```

Figure 5.2: ASD structural representation of calculateService1



```
Aug 5, 2014 12:42:38 AM org.apache.commons.httpclient
WARNING: Going to buffer response body of large or un
Aug 5, 2014 12:42:38 AM org.apache.commons.httpclient
WARNING: Going to buffer response body of large or un
CCF assessment is executed:
two calculateServices are compatible
the first service version: 1.0
the second service version is calculated: 1.1
```

Figure 5.3: Result of CCF (calculateService1.0, calculateService1.1)



```
Aug 5, 2014 12:45:28 AM org.apache.commons.httpcl
WARNING: Going to buffer response body of large o
Aug 5, 2014 12:45:28 AM org.apache.commons.httpcl
WARNING: Going to buffer response body of large o
CCF assessment is executed:
two calculateServices are incompatible
the first service version: 1.2
the second service version is calculated: 2.0
```

Figure 5.4: Result of CCF (calculateService1.2, calculateService2.0)

Because the output-type of calculateService4 "double" is not subtype of "integer" which is the output-type of calculateService3, these two services are evaluated in the CCF assessment as "incompatible" with each other. Therefore, the major number is increased by 1 and so the version of calculateService4 becomes 2.0 (see Figure 5.4).

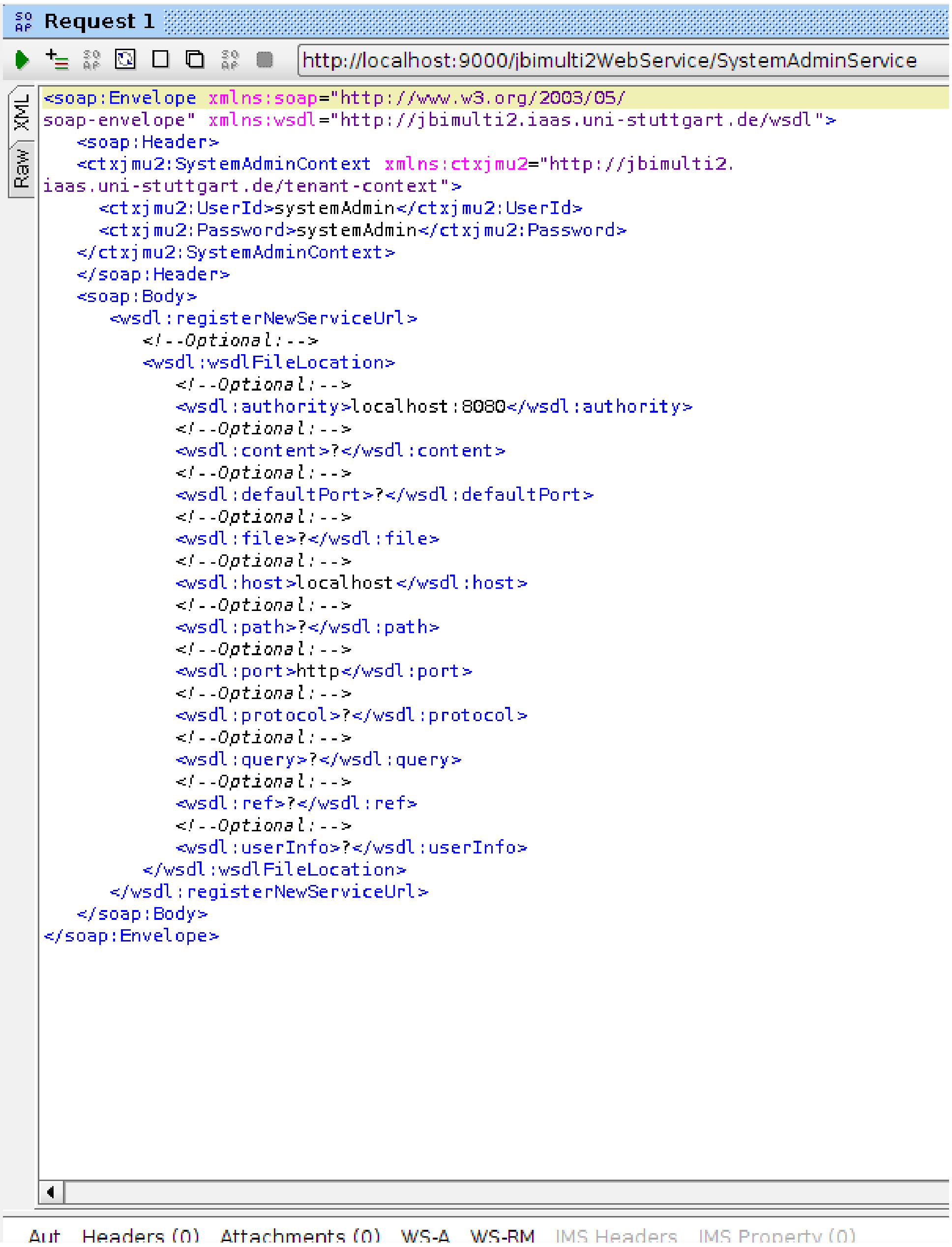
CCF (serviceA, serviceB)	Result of CCF assessment	Result of guild-line based approach
calculateService1 calculateService2	compatible (1.0, 1.1)	compatible (1.0, 1.1)
calculateService2 calculateService3	compatible (1.1, 1.2)	compatible (1.1, 1.2)
calculateService3 calculateService4	incompatible (1.2, 2.0)	incompatible (1.2, 2.0)
calculateService4 calculateService5	compatible (2.0, 2.1)	compatible (2.0, 2.1)
calculateService4 calculateService5	incompatible (2.0, 3.0)	incompatible (2.0, 3.0)

Table 5.6: Result of CCF (serviceA, serviceB)

Table 5.6 compares the results of CCF assessment with the deduced results. Obviously they are consistent.

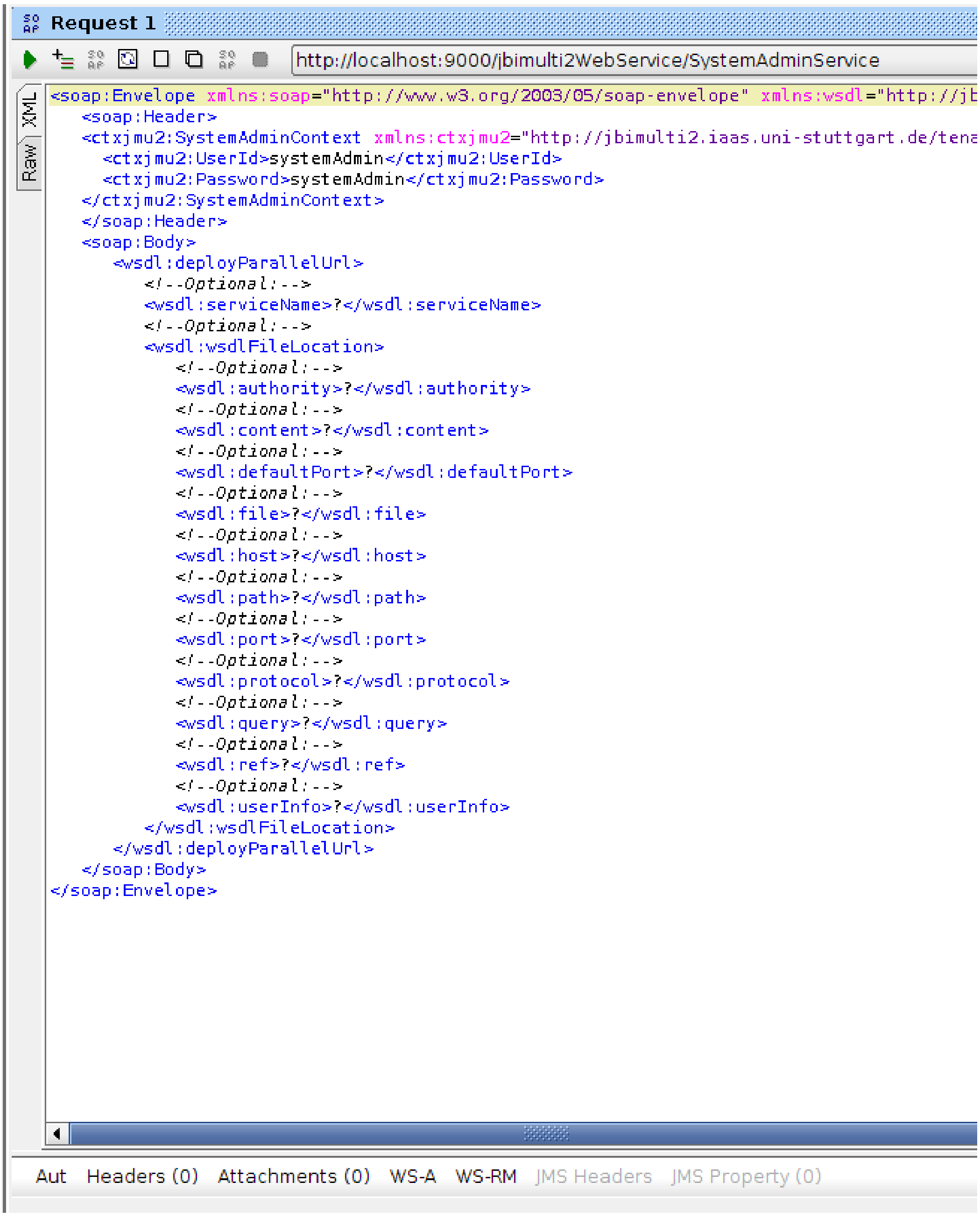
5.3 Validation of Versioned Service Control Service

The *registerNewService*, *replaceService* and *deployParallel* are exposed as Web Services provided by the JBIMulti2 system to manage service evolution. SoapUI 3.6 is used to interact with JBIMulti2 over the Web Service API. Figures 5.5, 5.6 and 5.7 show the requests of *registerNewService*, *replaceService* and *deployParallel* respectively. In each request message, the user identifier and password are set as *systemAdmin* for a system administrator. All parameters contained by *<wsdl: wsdlFileLocation>* define together where a new service version is deployed. *<wsdl: versionedServiceName>* in Figure 5.6 denotes the service name of an old service version which is replaced by the new service version. *<wsdl: serviceName>* in Figure 5.7 is the service name of an existing service version with which the new service version is deployed.



```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/
soap-envelope" xmlns:wsdl="http://jbimulti2.iaas.uni-stuttgart.de/wsdl">
   <soap:Header>
   <ctxjmu2:SystemAdminContext xmlns:ctxjmu2="http://jbimulti2.
iaas.uni-stuttgart.de/tenant-context">
     <ctxjmu2:UserId>systemAdmin</ctxjmu2:UserId>
     <ctxjmu2:Password>systemAdmin</ctxjmu2:Password>
   </ctxjmu2:SystemAdminContext>
   </soap:Header>
   <soap:Body>
      <wsdl:registerNewServiceUrl>
         <!--Optional:-->
         <wsdl:wsdlFileLocation>
            <!--Optional:-->
            <wsdl:authority>localhost:8080</wsdl:authority>
            <!--Optional:-->
            <wsdl:content>?</wsdl:content>
            <!--Optional:-->
            <wsdl:defaultPort>?</wsdl:defaultPort>
            <!--Optional:-->
            <wsdl:file>?</wsdl:file>
            <!--Optional:-->
            <wsdl:host>localhost</wsdl:host>
            <!--Optional:-->
            <wsdl:path>?</wsdl:path>
            <!--Optional:-->
            <wsdl:port>http</wsdl:port>
            <!--Optional:-->
            <wsdl:protocol>?</wsdl:protocol>
            <!--Optional:-->
            <wsdl:query>?</wsdl:query>
            <!--Optional:-->
            <wsdl:ref>?</wsdl:ref>
            <!--Optional:-->
            <wsdl:userInfo>?</wsdl:userInfo>
         </wsdl:wsdlFileLocation>
      </wsdl:registerNewServiceUrl>
   </soap:Body>
</soap:Envelope>
```

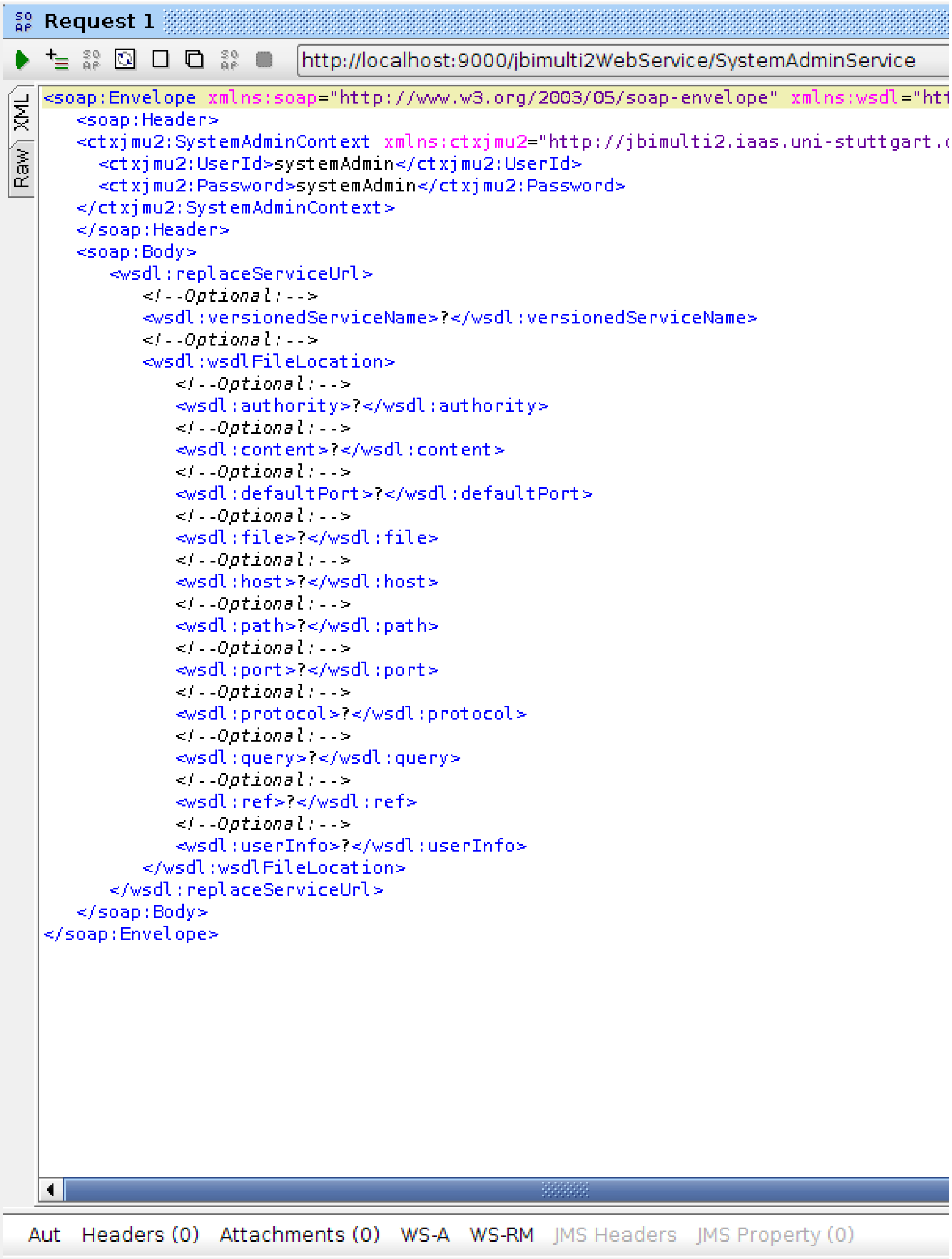
Figure 5.5: Register New Service Request executed by system administrator with soapUI 3.6



http://localhost:9000/jbimulti2WebService/SystemAdminService

```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope" xmlns:wsdl="http://jb
   <soap:Header>
   <ctxjmu2:SystemAdminContext xmlns:ctxjmu2="http://jbimulti2.iaas.uni-stuttgart.de/tena
     <ctxjmu2:UserId>systemAdmin</ctxjmu2:UserId>
     <ctxjmu2:Password>systemAdmin</ctxjmu2:Password>
   </ctxjmu2:SystemAdminContext>
   </soap:Header>
   <soap:Body>
      <wsdl:deployParallelUrl>
         <!--Optional:-->
         <wsdl:serviceName>?</wsdl:serviceName>
         <!--Optional:-->
         <wsdl:wsdlFileLocation>
            <!--Optional:-->
            <wsdl:authority>?</wsdl:authority>
            <!--Optional:-->
            <wsdl:content>?</wsdl:content>
            <!--Optional:-->
            <wsdl:defaultPort>?</wsdl:defaultPort>
            <!--Optional:-->
            <wsdl:file>?</wsdl:file>
            <!--Optional:-->
            <wsdl:host>?</wsdl:host>
            <!--Optional:-->
            <wsdl:path>?</wsdl:path>
            <!--Optional:-->
            <wsdl:port>?</wsdl:port>
            <!--Optional:-->
            <wsdl:protocol>?</wsdl:protocol>
            <!--Optional:-->
            <wsdl:query>?</wsdl:query>
            <!--Optional:-->
            <wsdl:ref>?</wsdl:ref>
            <!--Optional:-->
            <wsdl:userInfo>?</wsdl:userInfo>
         </wsdl:wsdlFileLocation>
      </wsdl:deployParallelUrl>
   </soap:Body>
</soap:Envelope>
```

Figure 5.6: Replace Service Request executed by system administrator with soapUI 3.6



http://localhost:9000/jbimulti2WebService/SystemAdminService

```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope" xmlns:wsdl="htt
   <soap:Header>
   <ctxjmu2:SystemAdminContext xmlns:ctxjmu2="http://jbimulti2.iaas.uni-stuttgart.
     <ctxjmu2:UserId>systemAdmin</ctxjmu2:UserId>
     <ctxjmu2:Password>systemAdmin</ctxjmu2:Password>
   </ctxjmu2:SystemAdminContext>
   </soap:Header>
   <soap:Body>
      <wsdl:replaceServiceUrl>
         <!--Optional:-->
         <wsdl:versionedServiceName>?</wsdl:versionedServiceName>
         <!--Optional:-->
         <wsdl:wsdlFileLocation>
            <!--Optional:-->
            <wsdl:authority>?</wsdl:authority>
            <!--Optional:-->
            <wsdl:content>?</wsdl:content>
            <!--Optional:-->
            <wsdl:defaultPort>?</wsdl:defaultPort>
            <!--Optional:-->
            <wsdl:file>?</wsdl:file>
            <!--Optional:-->
            <wsdl:host>?</wsdl:host>
            <!--Optional:-->
            <wsdl:path>?</wsdl:path>
            <!--Optional:-->
            <wsdl:port>?</wsdl:port>
            <!--Optional:-->
            <wsdl:protocol>?</wsdl:protocol>
            <!--Optional:-->
            <wsdl:query>?</wsdl:query>
            <!--Optional:-->
            <wsdl:ref>?</wsdl:ref>
            <!--Optional:-->
            <wsdl:userInfo>?</wsdl:userInfo>
         </wsdl:wsdlFileLocation>
      </wsdl:replaceServiceUrl>
   </soap:Body>
</soap:Envelope>
```

Figure 5.7: Deploy Service in Parallel Request executed by system administrator with soapUI 3.6

6 Conclusions

Service evolution is a continuous process of maintenance and improvement of services [Pap08]. It enables a new service to be deployed, an old service to be decommissioned and multiple services to be available at the same time. An inefficient evolution management could result in breaks between service consumers and providers. This diploma thesis has focused on extending a multi-tenant aware ESB solution with service evolution management based on the JBIMulti2 system developed by Dominik Muhler [Muh12].

In Chapter 2 we have provided the needed fundamentals such as cloud computing, SOA, ESB, JBI and Apache ServiceMix, as well as introducing the essential works on which this thesis depends: ESB^{MT}, JBIMulti2, an overview of service evolution based on [ABP12], service version management framework and an existing extension design of the JBIMulti2 system for service version management. After deep investigations of the related works, we have analyzed in Chapter 3 the requirements of service evolution management and applied three use cases for service version control management. The CCF assessment has been added into those use cases to check whether two operated service versions are compatible. A Version Registry has been used to store all version-related information, such as the version identifier and the compatibility status. Furthermore, we discussed how to extend the SEs of an ESB instance by adding message interceptors and transformers, with the purpose of providing correct routing between service consumers and providers. Chapter 4 has described the technologies that are necessary to implement the service version control management system. The Version Registry has been implemented as a database at the source layer. The formal representation of service descriptions has been realized by analyzing XML-based WSDL files and it has combined with the CCF Algorithm proposed by [ABP12] to perform the compatibility checking between two service versions and the calculation of service version identifiers. The use cases discussed in Section 3.2 have been implemented as the functionalities provided by the JBIMulti2 system to manage the service evolution. In Chapter 5, we have designed a calculation service with multiple versions to validate the Compatibility Checking Function. The results of the CCF assessment are consistent with the guideline-based approach. We have also employed soapUI 3.6 to validate the service evolution management services provided by the JBIMulti2 system.

In this thesis we have extended the JBIMulti2 system to manage the service evolution. Services provided by the JBIMulti2 system such as *registerNewService*, *replaceService*, and *deployParallel*, as well as the CCF assessment between two services have been implemented. Moreover, in Section 3.3.5, we have designed a message interceptor and a message transformer to ensure the dynamic routing of messages. The message interceptor is responsible for intercepting each incoming message sent to SEs and determining its real target by accessing the Version Registry and invoking the transformer. The message transformer is able to convert the incoming message into the corresponding outgoing message needed by message receivers. Both of them could be realized in the future. Last but not least, the CCF assessment could be extended to support behavioral and non-functional layers of ASD meta-model [ABP12]. The service version control management system should be further improved and completed.

Bibliography

[4Ca] 4CaaSt. The 4CaaSt project. http://www.4caast.eu. (Cited on page 22)

[ABP08] Vasilios Andrikopoulos, Salima Benbernou, and Mike P Papazoglou. Managing the evolution of service specifications. In *Advanced Information Systems Engineering*, pages 359–374. Springer, 2008. (Cited on pages 29 and 48)

[ABP12] Vasilios Andrikopoulos, Salima Benbernou, and Michael P Papazoglou. On the evolution of services. *Software Engineering, IEEE Transactions on*, 38(3):609–628, 2012. (Cited on pages 6, 7, 8, 10, 26, 27, 28, 29, 30, 48, 75 and 76)

[All11] OSGi Alliance. OSGi Service Platform: Core Specification Version 4.3, 2011, 2011. http://www.osgi.org/Download/Release4V43/. (Cited on page 21)

[And10] Vasilios Andrikopoulos. A theory and model for the evolution of software services. Technical report, Tilburg University, 2010. (Cited on page 28)

[Arc] Service Architecture. Definition of SOA. www.service-architecture.com/articles/web-services/service-oriented_architecture_soa_definition.html. (Cited on page 17)

[Bar09] Neil Bartlett. OSGi in practice. *Bd (January 11, 2009)*, 2009. https://s3.amazonaws.com/neilbartlett.name/osgibook_preview_20090110.pdf. (Cited on page 21)

[BE04] Kyle Brown and Michael Ellis. Best practices for Web services versioning. *IBM developerWorks*, January 2004. http://www.ibm.com/developerworks/webservices/library/ws-version/. (Cited on page 30)

[BR00] Keith H Bennett and Václav T Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 73–87. ACM, 2000. (Cited on page 26)

[Bus] The Open Enterprise Service Bus. http://www.open-esb.net/. (Cited on page 19)

[Cha04] David Chappell. *Enterprise service bus.* " O'Reilly Media, Inc.", 2004. (Cited on page 18)

[Com] MuleSoft Community. http://www.mulesoft.org/. (Cited on page 19)

[Con] OW2 Consortium. JOnAS: Java Open Application Server. http://wiki.jonas.ow2.org. (Cited on page 68)

[Cor12] FuseSource Corp. Fuse ESB Enterprise: Using Java Business Integration, July 2012. http://fusesource.com/docs/esbent/7.0/jbi/jbi.pdf. (Cited on pages 6, 19 and 20)

[FJB10] Debbie Carson Ian Evans Scott Fordin, Kim Haase Eric Jendrock, and Jennifer Ball. The Java EE 5 Tutorial, 2010. http://docs.oracle.com/javaee/5/tutorial/doc/bnaax.html. (Cited on page 58)

[FLF$^+$07] Ru Fang, Linh Lam, Liana Fong, David Frank, Christopher Vignola, Ying Chen, and Nan Du. A version-aware approach for web service directory. In *Web Services, 2007. ICWS 2007. IEEE International Conference on*, pages 406–413. IEEE, July 2007. (Cited on page 30)

[Foua] The Apache Software Foundation. Apache ActiveMQ. http://activemq.apache.org/. (Cited on page 22)

[Foub] The Apache Software Foundation. Apache Maven. http://maven.apache.org/. (Cited on pages 22 and 59)

[Fouc] The Apache Software Foundation. Apache ServiceMix. http://servicemix.apache.org/. (Cited on pages 19, 21 and 59)

[Foud] The Apache Software Foundation. Apache Tomcat. http://tomcat.apache.org/. (Cited on page 68)

[Fou11a] The Apache Software Foundation. Apache Camel User Guide 2.7.0, 2011. http://camel.apache.org/manual/camel-manual-2.7.0.pdf. (Cited on page 22)

[Fou11b] The Apache Software Foundation. Apache Karaf Users' Guide 2.2.5, 2011. http://repo1.maven.org/maven2/org/apache/karaf/manual/2.2.5/manual-2.2.5.pdf. (Cited on page 21)

[Gro] The Open Group. The SOA Work Group: Definition of SOA. http://www.opengroup.org/soa/source-book/soa/soa.htm. (Cited on page 17)

[JD08] K Jerijærvi and J Dubray. Contract versioning, compatibility and composability. *InfoQ Magazine*, 2008. http://www.infoq.com/articles/contract-versioning-comp2. (Cited on pages 26 and 30)

[JSBR09] Matjaz B Juric, Ana Sasa, Bostjan Brumen, and Ivan Rozman. WSDL and UDDI extensions for version support in web services. *Journal of Systems and Software*, 82(8):1326–1343, 2009. (Cited on page 30)

[Lie13] Sumadi Lie. Enabling the Compatible Evolution of Services based on a Cloud-enabled ESB Solution. Diplomarbeit, University of Stuttgart, Institute of Architecture of Application Systems, 2013. (Cited on pages 6, 30, 31 and 39)

[MG11] Peter Mell and Tim Grance. The NIST definition of cloud computing. 2011. (Cited on pages 15 and 16)

[Muh12] Dominik Muhler. Extending an Open Source Enterprise Service Bus for Multi-Tenancy Support Focusing on Administration and Management. Diplomarbeit, University of Stuttgart, Institute of Architecture of Application Systems, 2012. (Cited on pages 6, 9, 23, 24, 25, 35, 39 and 75)

[NS07] A Narayan and I Singh. Designing and versioning compatible Web services. *IBM DeveloperWorks*, 28, March 2007. http://www.ibm.com/developerworks/websphere/library/techarticles/0705_narayan/0705_narayan.html. (Cited on page 30)

[Ort07] Sixto Ortiz. Getting on board the enterprise service bus. *Computer*, 40(4):15–17, 2007. (Cited on page 18)

[PAB11] Michael P Papazoglou, Vasilios Andrikopoulos, and Salima Benbernou. Managing evolving services. *Software, IEEE*, 28(3):49–55, 2011. (Cited on pages 25 and 26)

[Pap08] Mike P Papazoglou. The challenges of service evolution. In *Advanced Information Systems Engineering*, pages 1–15. Springer, 2008. (Cited on pages 25 and 75)

[PAS04] Chris Peltz and Anjali Anagol-Subbarao. Design Strategies for Web Services Versioning. SYS-CON Media, Inc., April 2004. http://soa.sys-con.com/node/44356. (Cited on page 26)

[Pos] PostgreSQL. http://www.postgresql.org/. (Cited on page 58)

[PTDL07] MP Parazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service oriented computing: State of the art and research challenges. *Computer*, 40(11):38–45, 2007. (Cited on pages 17 and 18)

[RB14] B Rajmohan and M Balashankar. The Cloud and SOA – Creating the Architecture for Today and Future. *International Journal of Research in Engineering & Advanced Technology*, 1,Issue 6, Dec-Jan 2014. http://www.ijreat.org/Papers%202013/Issue5/IJREATV1I6035.pdf. (Cited on page 16)

[RD08] Tijs Rademakers and Jos Dirksen. *Open-source ESBs in action.* Manning, 2008. (Cited on page 18)

[SAGSL13] Steve Strauch, Vasilios Andrikopoulos, Santiago Gómez Sáez, and F Leymann. ESBMT: A Multi-tenant Aware Enterprise Service Bus. *International Journal of Next-Generation Computing*, 4(3):230–249, 2013. (Cited on pages 22 and 23)

[SAL$^+$12] Steve Strauch, Vasilios Andrikopoulos, Frank Leymann, Dominik Muhler, et al. ESBMT: Enabling Multi-Tenancy in Enterprise Service Buses. In *CloudCom*, pages 456–463. Citeseer, 2012. (Cited on pages 22, 23 and 24)

[SCFY96] Ravi S Sandhu, Edward J Coyne, Hal L Feinstein, and Charles E Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996. (Cited on page 34)

[Sof] SmartBear Software. soapUI. http://www.soapui.org/. (Cited on page 68)

[THW05] R Ten-Hove and P Walker. JavaTM Business Integration (JBI) 1.0-JSR 208 Final Release. *Sun Microsystems, Inc., Tech. Rep*, 2005. (Cited on page 20)

[WCL$^+$05] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald F Ferguson. *Web services platform architecture: SOAP, WSDL, WS-policy, WS-addressing, WS-BPEL, WS-reliable messaging and more.* Prentice Hall PTR, 2005. (Cited on page 17)

All links were last followed on August 14, 2014.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Stuttgart, August 15. 2014, Penghao Tian