University of Plymouth

PEARL **https://pearl.plymouth.ac.uk**

04 University of Plymouth Research Theses 01 Research Theses Main Collection

2015

Automated Creation and Provisioning of Value-added Telecommunication Services

Eichelmann, Thomas

http://hdl.handle.net/10026.1/3376

http://dx.doi.org/10.24382/1493

Plymouth University

All content in PEARL is protected by copyright law. Author manuscripts are made available in accordance with publisher policies. Please cite only the published version using the details provided on the item record or document. In the absence of an open licence (e.g. Creative Commons), permissions for further reuse of content should be sought from the publisher or author.

Copyright Statement

This copy of the thesis has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without the author's prior consent.

Automated Creation and Provisioning of Value-added Telecommunication Services

by

Thomas Eichelmann

A thesis submitted to Plymouth University

in partial fulfilment for the degree of

Doctor of Philosophy

School of Computing and Mathematics

In collaboration with

Darmstadt Node of the CSCAN Network

April 2015

Automated Creation and Provisioning of Value-added Telecommunication Services

Thomas Eichelmann

Abstract

The subject of this research is to find a continuous solution, which allows the description, the creation, the provisioning, and the execution of value-added telecommunication services. This work proposes a framework for an easy and timesaving creation and provisioning of value-added telecommunication services in Next Generation Networks.

As research method, feasibility, comparative methods are used in this study. Criteria and requirements for service description, service creation, service execution, and service provisioning, are defined and existing technologies are compared with each other and evaluated regarding these criteria and requirements. Extensions to the selected technologies are proposed and possibilities to combine these technologies are researched. From the results of the previous steps, a framework is defined which offers a continuous solution for the description, creation, provisioning and execution of value-added services. In order to test the proof of concept, this framework is prototypically implemented. For a qualitative analysis of the research targets and the proof of concept, an example service is created and executed within the framework prototype. Furthermore, in order to examine the validity of the quantitative aims and objectives of this research work, a second example service is created, and its characteristics are measured and analysed.

The result of this research is a novel continuous approach for the creation of value-added telecommunication services. This research introduces new possibilities for the service description, service creation, service provisioning, and service execution through an extension of the common telecommunication real-time execution environment JAIN SLEE. Value-added services are described by using the business process execution language BPEL. This language facilitates a simple and fast service design. The service can automatically be composed from pre-defined and pre-deployed components.

Contents

List of Figures

List of Tables

Acknowledgements

The research behind this dissertation could not have been accomplished without the personal and practical support of numerous people. It would be impossible to thank all of those who helped and contributed somehow to the development of this work. I would like to acknowledge the contributions of the following people:

- My supervisors Prof. Dr. Woldemar Fuhrmann, Prof. Dr.-Ing. Ulrich Trick and Dr. Bogdan V. Ghita for their professional advice, personal support, guidance in research, and their encouragements throughout the project.

- My colleagues from the Research Group for their friendship and for their help, whenever requested.

- My family and friends, for their support and encouragements, without which this thesis would never have been written.

The research projects TeamCom and ComGeneration providing the basis for this publication were partially funded by the Federal Ministry of Education and Research (BMBF) of the Federal Republic of Germany under grant number 1704B07 and 1724B09. The authors of this publication are in charge of its content.

Author’s Declaration

At no time during the registration for the degree of Doctor of Philosophy has the author been registered for any other University award without prior agreement of the Graduate Committee.

Relevant scientific seminars and conferences were regularly attended at which work was often presented, and several papers were prepared for publication.

Word count of main body of thesis: 70.125

Signed ________________________

Date ________________________

1 Introduction

Service providers have to react fast to market changes and emerging trends in order to respond to market needs, remain competitive, and offer new services whenever requested by the customers. Typically, the development of new services is currently still very time-consuming and would require weeks or months to develop a new value-added service.

In the past, telephony comprised simple audio connections between two participants. With the new emerging possibilities like video conferencing, instant messaging, presence or web applications, value-added services have broadened their scope. Furthermore, today's services also need to support new resources and must be able to integrate new protocols.

Multiple media resources can be combined into one service and, furthermore, a service can support multiple protocols, for example, an audio conference service with translation. Multiple participants can join this conference service, and for each participant the voice communication is translated into the requested target language. In this service, a signalling protocol is required to establish the connections with the conference service and the participants. Moreover, a protocol is required to transmit the user data from the end-user equipment to a media server that is controlled by the service. This media server can apply voice recognition to the received user data, translates the voice communication into text, and forwards the generated text to, e.g., a translation web service. The web service translates the text into the required languages and sends the translated text to the media server. The media service then translates the text into speech and streams it to the user.

Currently, the development of value-added services requires a lot of detailed knowledge about the communication systems and the associated protocols; lacking the abstraction level and remaining incompatible with mainstream web application development. Therefore, specialists with a broad technical knowledge are required for the development process. The required deep skills hinder the expansion of value-added services, only the telecommunication industry is able to develop value-added services, which are costly as well as time and resource consuming. This thesis offers a novel solution for an easy, graphical description of value-added telecommunication services and an automated service creation from the service description. The value-added telecommunication services are executed in an extended service execution environment.

The detailed aims and objectives of this research are presented in section 1.1, followed by an outline of the thesis structure in section 1.2.

1.1 Aims and Objectives

This thesis presents a novel solution for the development of value-added services. A value-added service is described first on a logical level using well-known formal methods used in information technology (IT). The service description is automatically compiled into a program that can be executed in an appropriate telecommunication real-time runtime environment. Communication Building Blocks (CBBs) are created to provide the required resources for the formal value-added service description and implement the technical functionalities in the execution environment.

This research work aims to find and describe a novel, approach for the creation and provisioning of value-added telecommunication services, which spans the description, creation, provisioning, and execution of a service. The proposed approach uses a higher abstraction level comparable with web application development in the context of service-oriented architectures and reusable/composable building blocks, derives the service from the service description, and uses predefined service components to represent it. The generated value-added service is provided and executed by a telecommunication framework.

The service creation environment supports the application developer in designing logical value-added services. The functionality of the service and the protocol support are based on reusable components called “communication building blocks” (CBBs). The CBBs are mapping the description of the functionality to the implementation of the functionality in the service execution environment (SEE).

Within the proposed framework, the service description stage is implemented using the Business Process Execution Language (BPEL) (OASIS, 2007). This language was created and optimised to allow for an easy formal definition and description of business processes. BPEL, however, was not developed for controlling real-time communication services in heterogeneous networks. Therefore, BPEL is only used for the description of a service, not for execution. Furthermore, a mechanism is proposed to automatically generate the value-added service implementation from the service description. This mechanism maps the logic elements defined in BPEL to service components in a Service Execution Environment (SEE) and composes these components to the value-added service.

The SEE is based on JAIN SLEE (JAIN Service Logic Execution Environment) (Sun and Open Cloud, 2008) and is enhanced by necessary extensions to support the requirements for this framework. The provided CBBs also offer an abstraction layer for underlying heterogeneous communication networks, allowing the developer to focus exclusively on the application logic, rather than dealing with the respective communication protocols. This leads to new opportunities for a rapid and efficient service creation using a new Service Creation Environment (SCE) with a customisable level of abstraction and automated service generation.

The main objectives of the thesis can be summarised as follows:

- To investigate the current solutions for service creation, service provisioning, and service execution regarding existing approaches for an automated solution for the creation and provisioning of value-added telecommunication services. Criteria are defined that have to be fulfilled by the desired technology. The existing technologies have to be analysed regarding the defined criteria.
- To derive a methodology for the description of the service logic and the description of the functionality to support an easy and fast development with graphical support.
- To propose and analyse service creation concepts based on a set of defined requirements.
- To design and implement a service execution environment based on a comprehensive and flexible environment. The resulting solution must link

directly the user defined requirements with the generated service through the service execution concepts.

- Based on the results of the previous research steps, the framework architecture and the structure of the services are derived. The result is a solution for the creation and provisioning of value-added telecommunication services.
- For the proof of concept and for the demonstration of the framework functionality, a prototype of the defined framework has to be implemented and tested.

The order of objectives declared above corresponds to the general structure of this thesis which will be presented in the following section.

1.2 Thesis Structure

Chapter 2 describes the theoretical background of this work, and especially focuses on the telecommunication infrastructure. In the first part of chapter 2, a more detailed description and definition of value-added services are given. Then, the required network capabilities are discussed, and the Internet, the Next Generation Networks (NGN), and the IMS are explained in a brief overview. The concept of a SIP application server for the execution of value-added services, as well as the concept of a media server which offers media handling for the services are explained. Furthermore, a general description of Service Delivery Platforms is presented. Finally, the criteria of service creation, service provisioning, and service execution are defined in order to be able to evaluate existing technologies and related research projects in chapter 3 and 4.

Chapter 3 introduces the current solutions for service creation. The limitations and weaknesses of these approaches are analysed, and related research projects are discussed. The technologies are evaluated regarding the criteria defined in section 2.5.

Chapter 4 discusses the problems associated with traditional service provisioning and execution. In this context, the existing approaches and related research projects with their limitations and weaknesses are investigated and evaluated regarding the criteria defined in section 2.5.

Chapter 5 uses the prior research presented in chapters 3 and 4 as a starting point for proposing a solution to address current limitations. In chapter 3, BPEL is selected as service description solution and in chapter 4; JAIN SLEE is selected as service execution solution. Chapter 5 describes new approaches, which cover these topics of service description, and service execution and propose solutions for service creation.

In chapter 6, the novel framework for automated creation and provisioning of value-added telecommunication services is proposed. This chapter uses the solutions and techniques from chapter 5 and starts with an architectural overview. Then, the proposed service creation and the service execution environment are analysed in more detail.

Chapter 7 introduces the structure and the life cycle of the services generated by the framework. The chapter starts with the description of the service structure, and explains the framework context, the variables, and variable types. The possibilities for internal and external service communication are described and the service components analysed. The last part of chapter 7 discusses the service life cycle with the defined states and the life cycle phases.

Chapter 8 provides the evaluation of the developed framework against the defined requirements and the proof of concept. There, the general layout and the architecture of the research prototype, as well as the relevant components of the prototype are described. The concept of this framework is proofed with the help of an example scenario.

Chapter 9 concludes the PhD thesis with a summary of the achievements of the work, an outline of the advantages of the proposed solution, a summary of the claims of novelty, a discussion of the limitations of the research, and potential avenues to pursue for future work.

2 Introducing the Telecommunication Infrastructure

This chapter describes the theoretical background of the telecommunication infrastructure. Criteria are defined in order to be able to evaluate the service description technologies and service execution technologies. The first section explains the concepts of services, service features and value-added services that are required for this thesis. Then, a brief overview of network-based service provisioning in the Internet, NGN, and IMS is given. In the next step, the concept of Application Server and Media Server is described. Moreover, the concept of Service Delivery Platforms is explained.

From the theoretical background and the aims and objectives declared above, criteria are derived for the evaluation of service description technologies (refer to chapter 3) and service execution technologies (refer to chapter 4).

2.1 Definition of Services

The term service is used in many domains, and there are different definitions of this term. This thesis concentrates on software components. In this context, the following definition is the most appropriate: the term service refers to the functionality offered by software components at defined interfaces. Services can be composed of service building blocks to form structured services and applications.

This thesis concentrates on value-added service of the telecommunications domain but for a better understanding of the technologies, described in the chapters 3 and 4; the Information Technology (IT) domain is shortly introduced in section 2.2.1. Therefore, in this section the definitions of the term IT, IT services, and of course, services in the telecommunications domain are presented and discussed. The first definition concentrates on the term "IT".

In the article "Evolution of SOA Concepts in Telecommunications" (Magedanz *et al.*, 2007), Thomas Magedanz, Niklas Blum, and Simon Dutkowski define the IT domain as follows:

"The Information technology is defined as computer communications, networks, and information systems that enable exchanges of digital objects. We can say also IT encompasses all forms of technology used to create, store, exchange, and use information in its various forms like business data, voice conversations, still images, motion pictures, multimedia presentations, etc." (Magedanz, 2007)

In the following step, the term IT service is defined.

ITILv3 (IT Infrastructure Library version 3) provides the following general definition of the term service within the IT:

"A Service is provided to one or more Customers, by an IT Service Provider. An IT Service is based on the use of Information Technology and supports the Customer's Business Process. An IT Service is made up from a combination of people, processes, and technology and should be defined in a Service Level Agreement." (ITIL, 2014)

"Service: A means of delivering value to customers by facilitating outcomes customers want to achieve without the ownership of specific costs and risks. The term 'service' is sometimes used as a synonym for core service, IT service or service package." (ITIL, 2014)

However, these are all-encompassing general definitions of the term IT service. This thesis is more concerned with services offered by software components. Familiar techniques which are common in the IT are service choreography and service orchestration. This thesis presents novel concepts based on these techniques in chapter 5, which is why the terms orchestration and choreography need to be clarified.

Matjaz B. Juric (Juric, 2014) defines orchestration as follows: "In orchestration, which is usually used in private business processes, a central process (which can be another web service) takes control of the involved web services and coordinates the execution of different operations on the web services involved in the operation. The involved web services do not 'know' (and do not need to know) that they are involved in a composition process and that they are taking part in a higher-level business process. Only the central coordinator of the orchestration is aware of this goal, so the orchestration is centralized with explicit definitions of operations and the order of invocation of web services". (Juric, 2014)

In the same article (Juric, 2014), Matjaz B. Juric gives the following definition of choreography: "Choreography, in contrast" to orchestration "does not rely on a central coordinator. Rather, each web service involved in the choreography knows exactly when to execute its operations and with whom to interact. Choreography is a collaborative effort focusing on the exchange of messages in public business processes. All participants in the choreography need to be aware of the business process, operations to execute, messages to exchange, and the timing of message exchanges.” (Juric 2014*)*.

Telecommunications “deals with capturing, processing, transmitting, and storing information”. In the telecommunication domain, there exist quite a number of definitions of the term service. Service definitions of the 3GPP (3rd Generation Partnership Project), ETSI (European Telecommunications Standards Institute) and ITU-T (International Telecommunication Union-Telecommunication Standardization Sector) provide the basis for the service definitions used in this thesis.

ETSI discriminate several service classes: (i) basic telecommunication services, which can be supplemented or modified by supplementary services, (ii) value-added non-call related services, and (iii) IP multimedia services. The general term for both basic bearer service and basic teleservice is basic telecommunication service (ETSI TS 122.105, 2008).

A basic bearer service is a type of telecommunication service that provides “the capability of transmission of signals between access points” (ETSI TS 122.105, 2008), i.e. capabilities of the OSI layers 1–3.

Basic teleservices are “telecommunication services providing the complete capability, including terminal equipment functions, for communication between users according to protocols established by agreement between network operators" (ETSI TS 122.105, 2008), i.e. capabilities of the OSI layers 1–7.

"A supplementary service modifies or supplements a basic telecommunication service. Consequently, it cannot be offered to a user as a stand-alone service. It shall be offered together or in association with a basic telecommunication service. The same supplementary service may be applicable to a number of basic telecommunication services" (ETSI TS 122.105, 2008).

To conclude: a telecommunication service (Table 2.1) is a combination of one or more bearer services and/or one or more teleservices. A telecommunication service can be modified and supplemented by one or more supplementary services (ETSI TS 122.001, 2009), (ITU-T I.210, 1993).

Table 2.1: Classification of telecommunication services (related to (ITU-T I.210, 1993))

telecommunication service	
teleservice	
basic teleservice	basic teleservice + supplementary service(s)
bearer service	
basic bearer service	basic bearer service + supplementary service(s)

ETSI distinguishes two further service classes, IP multimedia services, and value-added non-call related services.

A multimedia service (ETSI TS 122.101, 2009) combines “two or more media components (e.g. voice, audio, data, video, pictures) within one call. A multimedia service may involve several parties and connections (different parties may provide different media components) and therefore flexibility is required in order to add and delete both resources and parties” (ETSI TS 122.101, 2009). “Multimedia services are typically classified as interactive or distributed services” (ETSI TS 122.101, 2009). “IP multimedia services are the IP based session related services, including voice communications. IP multimedia sessions use IP bearer services provided by the PS CN” (packed switched core network) (ETSI TS 122.101, 2009).

“Value-added non-call related services include a large variety of different operator specific services/applications.” They do not need to be standardised. “The services can be based on fully proprietary protocols or standardised protocols” (ETSI TS 122.101, 2009). An overview of the described definitions is given in (Figure 2.1).

Another classification is put forward by the ITU-T. A service can be an interactive or a distribution service (ITU-T I.211, 1993). Interactive services can be classified as conversational, messaging, or retrieval services (ITU-T I.211, 1993). Distribution services can be categorised as distribution services without user-individual presentation control and distribution services with user individual presentation control (ITU-T I.211, 1993).

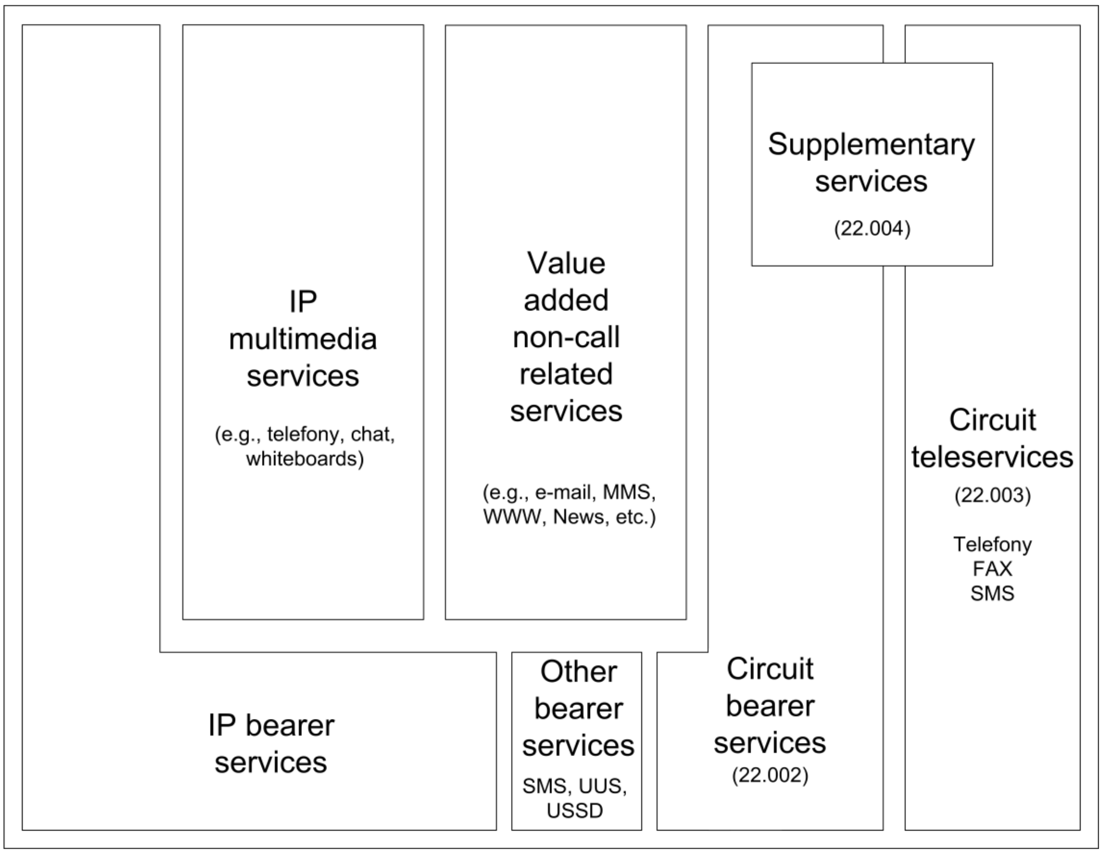


Figure 2.1: Service classification (related to (ETSI TS 122.101, 2009))

The following definition of the term value-added service is used in this research work. It is a combination of several definitions and is based on (Lehmann, 2010): "Value-added services are any functional properties that will offer a specific comfort and additional benefit to consumers of the services. Value-added services are based on a telecommunication service combining one or more bearer services (here solely IP bearer services), and/or one or more teleservices, and optionally, one or more supplementary services offered by a telecommunication operator. However, they are not services of the transport and call-control layers of the core network (refer to section 2.2.2). They also provide benefits that services of the transport and call-control layers cannot provide. Value-added services can be an add-on to basic services (bearer service and teleservice) and can sometimes be stand-alone operationally (e.g., non-call-related services)."

Since value-added services are not provided by the transport and call-control layers of the core network, additional network elements such as Application Servers (ASs) and Media Servers (MSs) are required. The AS provides a Service Execution Environment (SEE) for the value-added services. The MS processes and generates media streams. These network elements are described in section 2.3.

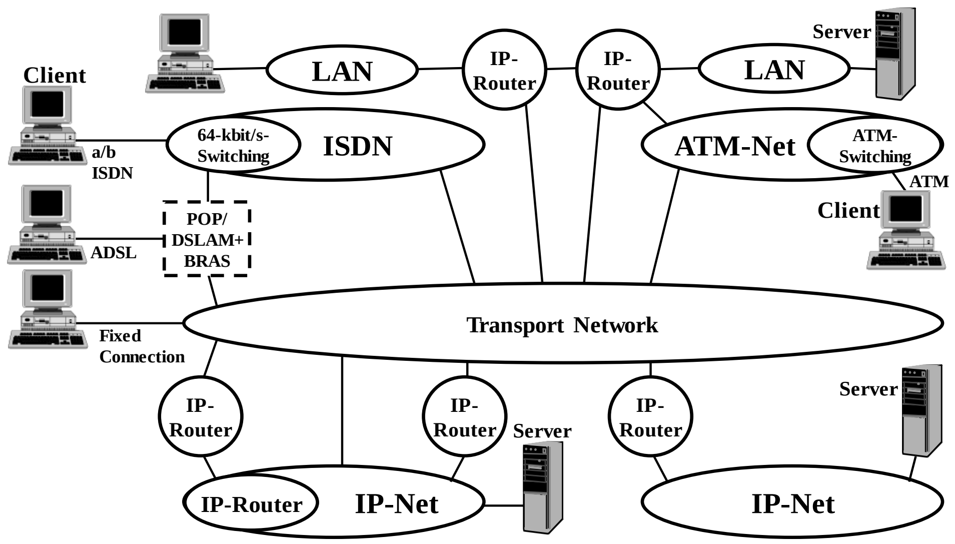
2.2 Network-based Service Provisioning

Since the term value-added service was described in the last section, this section shortly introduces the networks in which the value-added services operate. Within this section the characteristics of the Internet, NGN and IMS are discussed. This section is the basis for the service creation, provisioning, and execution technologies described in chapters 3 and 4.

2.2.1 Internet

The great advantage of the Internet is its capability of integrating various services, including multimedia services, and using IP as the underlying transport layer. It offers an open communication platform, which is globally available. Figure 2.2 shows the basic structure of the Internet. The Internet is an IP-based packet data network that is formed from subnets.

Despite the usage of underlying connection-based and circuit-switched networks, the communication on the Internet is connectionless and packet-switched. The routing is done based on IP addresses, where consecutive IP packets, despite having the same destination address, may take different routes. Until now, the Internet is working with best effort. All IP packets are forwarded by the router with the same priority, independent of the type of service. In summary, this implicates that the quality of service cannot be predicted. It is uncertain how much time a packet takes in the network, what is the amount of the jitter of a packet, and what is the probability that a packet gets lost. Therefore, the current Internet is very well suited for data services such as file transfer, e-mail, and web page requests but not for broad-band real-time services such as video conferencing or live TV (Trick and Weber, 2007).



POP = Point Of Presence
IP = Internet Protocol
ADSL = Asymmetric Digital Subscriber Line
ATM = Asynchronous Transfer Mode
LAN = Local Area Network
BRAS = Broadband Remote Access Server

Figure 2.2: The structure of the Internet (related to (Trick and Weber, 2009))

The Internet supports various standardised services. Services can use FTP (File Transfer Protocol), POP3 (Post Office Protocol version 3), HTTP (Hyper Text

Transfer Protocol), and remote management via Telnet (Telecommunication Network) as well as Internet telephony using SIP and Google Talk (Jabber/XMPP (Extensible Messaging and Presence Protocol)) (IETF RFC 6120, 2011). However, on the Internet also proprietary services like Skype and various other IM (Instant Messaging) programs are offered.

2.2.2 NGN (Next Generation Networks)

Next Generation Networks (NGN) are packet-oriented networks, which are able to provide telecommunication services. The general network structure of NGN is presented in Figure 2.3.

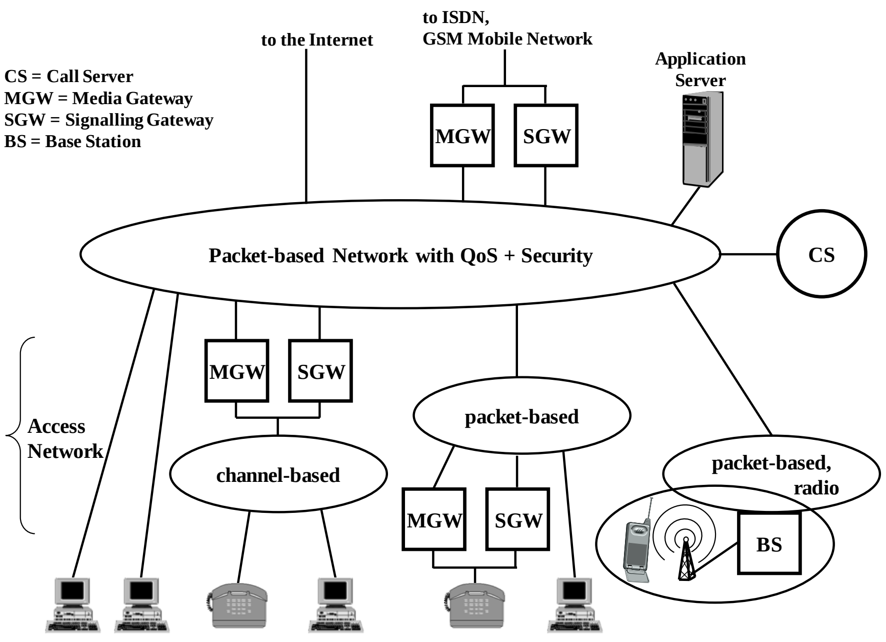


Figure 2.3: The architecture of a Next Generation Network (NGN) (related to (Trick and Weber, 2009))

The NGN can make use of multiple broadband, QoS-enabled transport technologies. Service-related functions are independent from underlying transport-related technologies. NGN enables unfettered access for users to networks and to competing service providers and/or services of their choice. It supports nomadic (non-seamless) mobility, which will allow consistent and ubiquitous provision of services to users (ITU-T Y.2001, 2004). NGN separates session control and service control layer from the transport layer. The use of IP allows the integration of heterogeneous transport technologies and supports open access for new services. It provides integrated security capabilities and it is compliant with specific regulatory requirements (e.g., emergency calls, lawful interception, security, privacy). For the provision of value-added services, application servers and media servers are used (refer to section 2.3).

Figure 2.4 presents the separation of the Call and Service Control Layer (Service Stratum) and the Application Layer (Application Stratum) from the Transport Layer (Transport Stratum).

With this separation in a strata/layer structure, a mapping from the service classes defined in section 2.1 to the NGN layers is possible. Bearer services can be mapped to the Transport Layer. Teleservices are executed in the Service Stratum whereby value-added services and possibly supplementary services are executed on the AS in the Application Stratum.

The interworking with legacy circuit-switched networks is provided by Signalling Gateways (SGWs) for the signalling and Media Gateways (MGWs) for the user data. These gateways are controlled by the Call Servers (CSs) or by Media Gateway Controllers (MGCs).

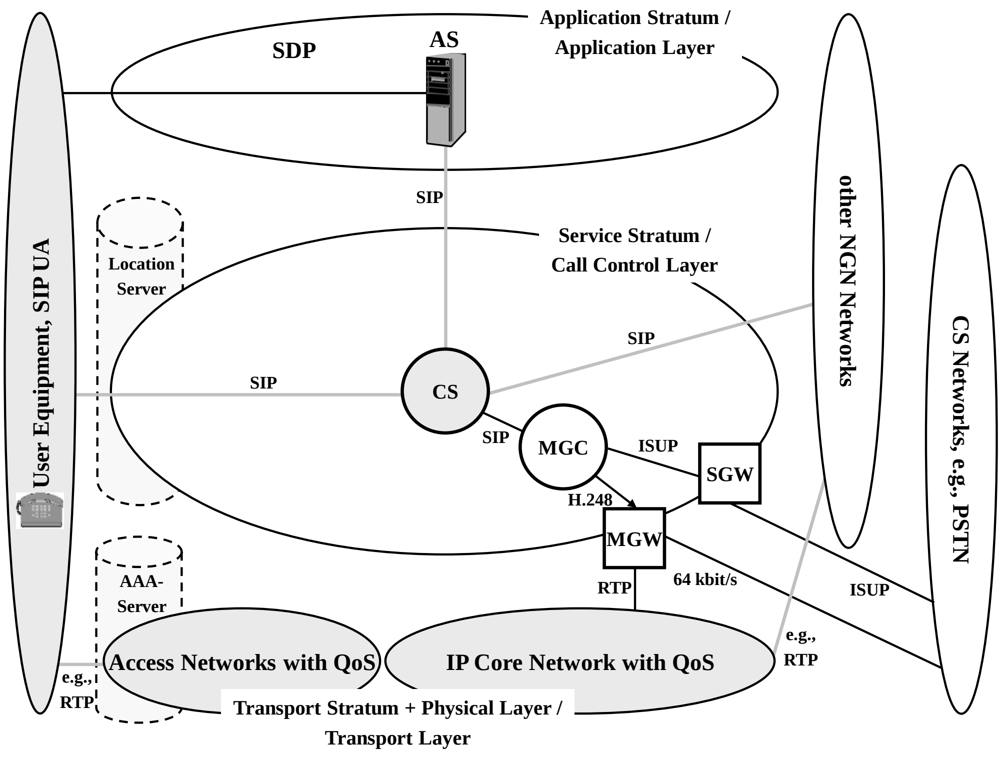


Figure 2.4: NGN architecture in a strata/layer structure (related to (Trick and Weber, 2009))

The NGN supports services with defined transport layers (TCP, UDP, TLS) and defined Quality of Service (QoS), which is important for real-time communication. For the provisioning of IP TV, high demands on the QoS are made, since image presentation is very sensitive to loss of data.

NGN support services with a defined QoS, a defined security protection for signalling and for user data, and they support the national regulatory requirements of the different countries.

2.2.3 IMS (IP Multimedia Subsystem)

IMS is a standardised control architecture to realise the session and service control in an NGN environment (3GPP TS 23.228, 2006) (ETSI TS 122.228, 2009). The IMS was initially standardised and implemented in the UMTS (Universal Mobile Telecommunications System) release 5 (3GPP TS 23.228, 2013).

IMS introduces new protocols (SIP, RTP, Megaco (Media Gateway Control Protocol), Diameter, etc.) and logical network elements (Call Session Control Function (CSCF) and Media Resource Function (MRF), etc.). Figure 2.5 presents the IMS architecture in a strata/layer structure.

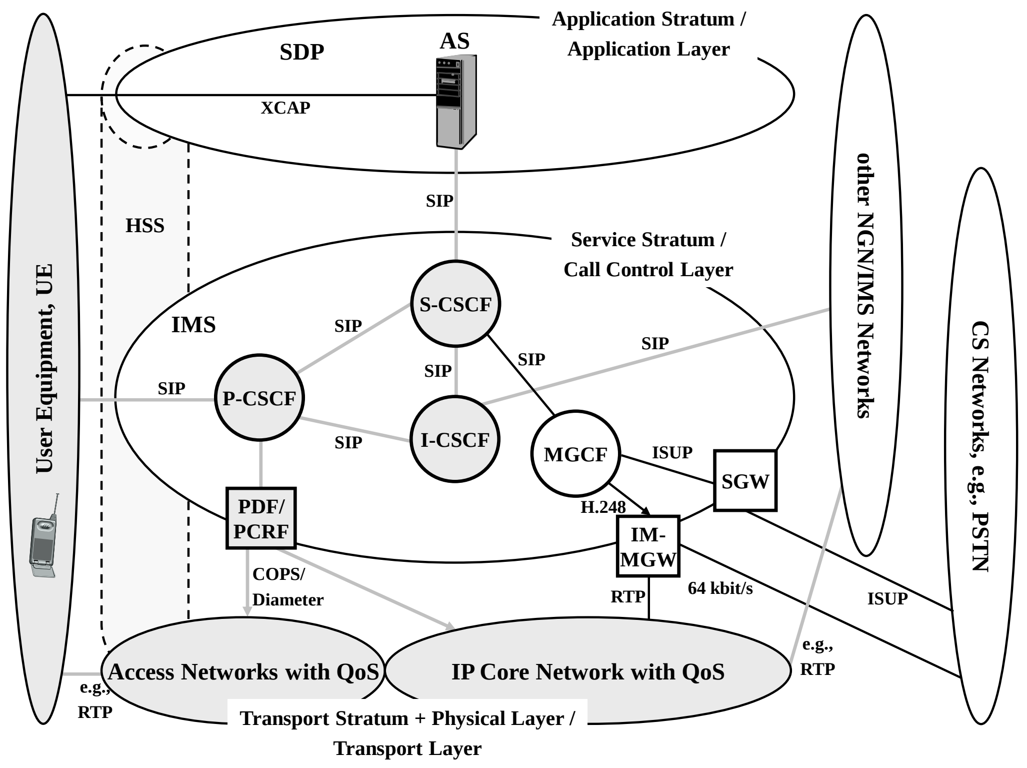


Figure 2.5: IMS architecture in a strata/layer structure (related to (Trick and Weber, 2009))

In contrast to the NGN structure in Figure 2.4 the Call Control Functions are represented by the CSCFs (Call Session Control Functions) in the Call Control Layer (3GPP, 2006). The Serving-Call Session Control Function (S-CSCF) corresponds to the Call Server (CS) in NGN. It is connected with the application servers that hosts and executes the services. Furthermore, the S-CSCF is supported by the optional

Interrogating-CSCFs (I-CSCFs), which select the responsible S-CSCF for incoming register or call requests in cooperation with the HSS. Furthermore, the I-CSCF hides the internal network structure from the outside. The Proxy-CSCFs (P-CSCFs) offer proxy functionality and serves as outbound proxy. Media Gateways (MGWs) and Signalling Gateways (SGWs) are used for the integration of other networks. The Media Gateway Control Function (MGCF) controls them. The Policy Decision Function (PDF) and the Policy and Charging Rules Function (PCRF) are monitoring and managing the QoS in the IP network.

In the IMS, the Media Server (refer to section 2.3) is called Media Resource Function (MRF). The MRF is subdivided into a MRFC (Media Resource Function Controller) and a MRFP (Media Resource Function Processor).

2.3 SIP Application Server and Media Server

In this section, the logical NGN network elements SIP AS (SIP Application Server) and MS (Media Server) are investigated in detail. The SIP AS provides the value-added services. The MS offers media handling functionality, which can be used by services running on the AS. Typical services, which use the media server functionalities, are video conferencing or IPTV.

A SIP AS consists of a software platform for services, a SIP Proxy, a Redirect Server, a SIP user agent, and/or a back-to-back user agent. The SIP AS can be realised as a stand-alone server or as an integrated call server. Application Servers enable a fast and cost-efficient provision of value-added services. (Trick *et al*. 2006)

To call a service on a SIP AS, SIP messages are routed through a Call Server (CS) to the SIP AS. The CS routes the SIP messages based on the configured or currently requested filtering criteria. On the base of further filter criteria, the AS decides about which service is executed. The service can be executed via application software like SIP servlets.

Possible modes of operation of a SIP AS (3GPP TS 23.228, 2006) are demonstrated in Figure 2.6. In the "Content" mode of operation, the SIP AS is used as SIP UA (User Agent) or redirect server. The UA of user A triggers the initiation of the service. User data is transmitted between the AS and the UA of user A. This mode can be used, e.g., to realise a voice box service.

In the "Wake-up" mode, the SIP AS acts as the initiator of the service. It represents a SIP UA contacting the UA of user B. User data is transmitted between the AS and the UA of user B. This mode can be used, e.g., to realise a Wake-up service.

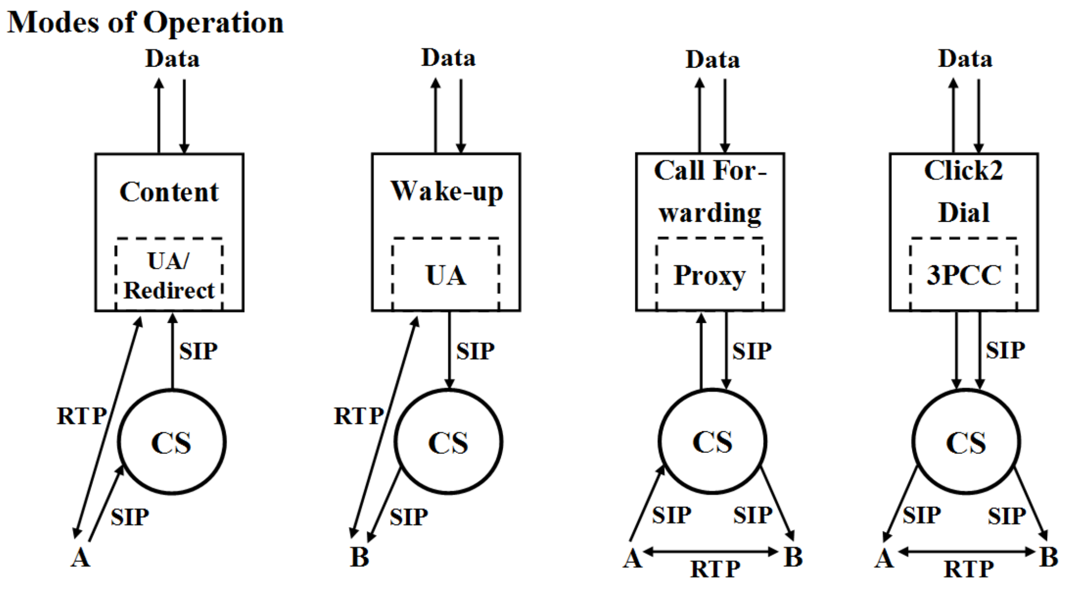


Figure 2.6: Modes of operation of an application server (related to (Trick and Weber, 2009))

The "Call Forwarding" mode shows a SIP AS with the function of a proxy. The UA of participant A contacts the UA of participant B. A sends a SIP message to the Call Server (CS). Because of its filter criteria – participant B, e.g., is unknown – the CS forwards the message to the SIP AS. The SIP AS determines the necessary data and provides the CS with the information. The call server is now able to forward the appropriate SIP message to participant B. After participant B confirmed the SIP message, both participants can exchange user data.

In the "Click2Dial" mode of operation, the SIP AS is acting as a B2BUA (Back-to-Back User Agent) to realise the Third-Party Call Control (3PCC) function. With the 3PCC function, the SIP AS arranges the session with the help of the CS between both parties. User data is transmitted directly between the UAs of user A and user B.

To implement the examples above, the SIP AS may require other servers such as e-mail servers, media servers, or web servers. In general, a service can make use of various functionalities provided by different servers. This principle is illustrated in Figure 2.7.

To provide the Click2Dial service, for instance, a web server with the corresponding web application is required by the 3PCC service on the SIP AS. The web application then triggers the described Click2Dial scenario on the SIP AS.

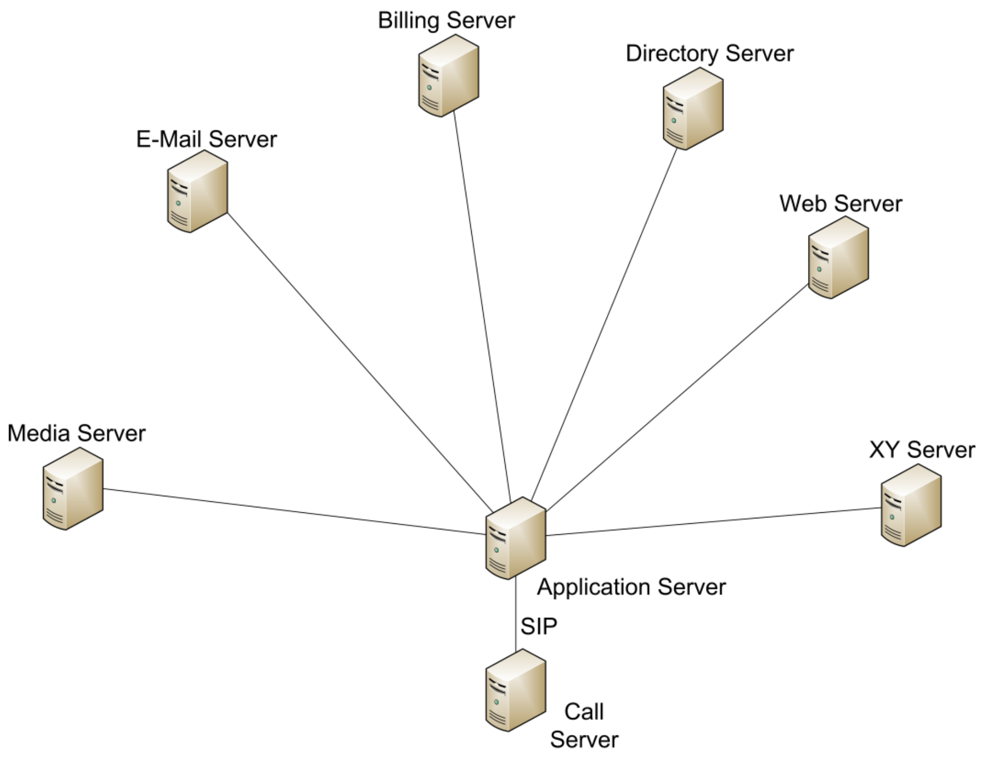


Figure 2.7: Possible environment of an SIP application server (related to (Trick and Weber, 2007))

A media server (MS) processes and generates media streams. The RTP protocol is usually used for the transmission of the media streams. The mixing of media data, transcoding between different media codecs, rescaling of videos, the interpretation of Dual-tone multi-frequency (DTMF) signalling, speech recognition (SR), text to speech (TTS), and media recording are some possible features of a media server. The MS is controlled by the services provided by the AS. Various protocols like VoiceXML (VXML) (W3C, 2007a), Media Server Control Mark-up Language (MSCML) (IETF RFC 5022, 2007), Media Server Mark-up Language (MSML) (IETF RFC 5707, 2010), and proprietary solutions has been developed for the communication between AS and MS.

AS and MS represent important logical network elements for the realisation of value-added services with multimedia user data. Figure 2.8 demonstrates how an AS can involve a MS for the handling of multimedia user data.

The figure shows a possible realisation of a SIP AS, which can be combined or not combined with a media server. The stand-alone SIP AS offers user data such as voice data by itself, thus the MS is integrated. Furthermore, it is possible that a SIP AS and a MS are integrated on the same computer. The MS then serves parts or all of the multimedia user data. In this case, a service on the SIP AS can control the media server through a proprietary interface. In the case of a non-combined media server, the network elements are physically separated, and the services on the AS control the MS via SIP and VoiceXML (IETF RFC 4267, 2005), SIP and MSCML (Media Server Control Markup Language) (IETF RFC 5022, 2007), or SIP and MSML (Media Server Markup Language) (IETF RFC 5707, 2010). (IETF, 2001a) and (IPCC, 2002)

The value-added services running on the AS are not required to implement the media processing, this is done by the MS. They control the media server by standardised or proprietary protocols.

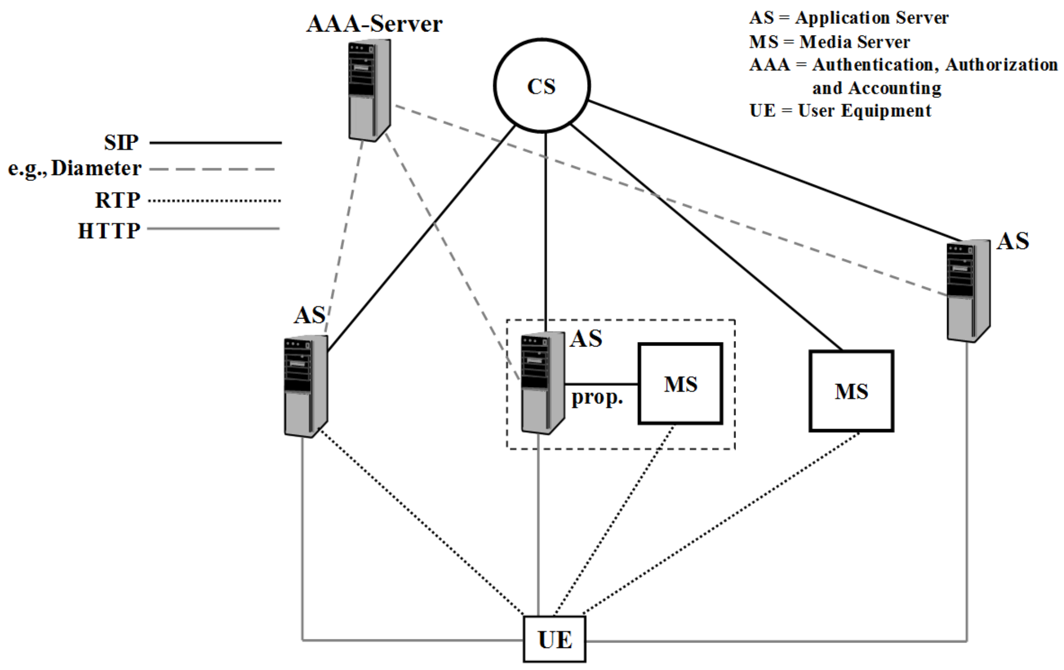


Figure 2.8: Interaction between the SIP AS and media server in NGN (related to (Trick and Weber, 2007))

2.4 Service Delivery Platform

For the provisioning of value-added services in next generation networks, Service Delivery Platforms are used by the providers. A Service Delivery Platform is a scalable platform for the creation, deployment, execution, orchestration, and management of value-added services. The Service Delivery Platform allows a service delivery across multiple types of networks, the creation of web services, IT services, and the usage of the providers' network capabilities. (Moriana, 2013)

"The term Service Delivery Platform refers to a system architecture or environment that enables the efficient creation, deployment, execution, orchestration, and management of one or more classes of services." (Moriana, 2013).

It is part of the application layer and offers an abstraction layer for the underlying protocols. An overview of the Service Delivery Platforms NGN integration is shown in Figure 2.9.

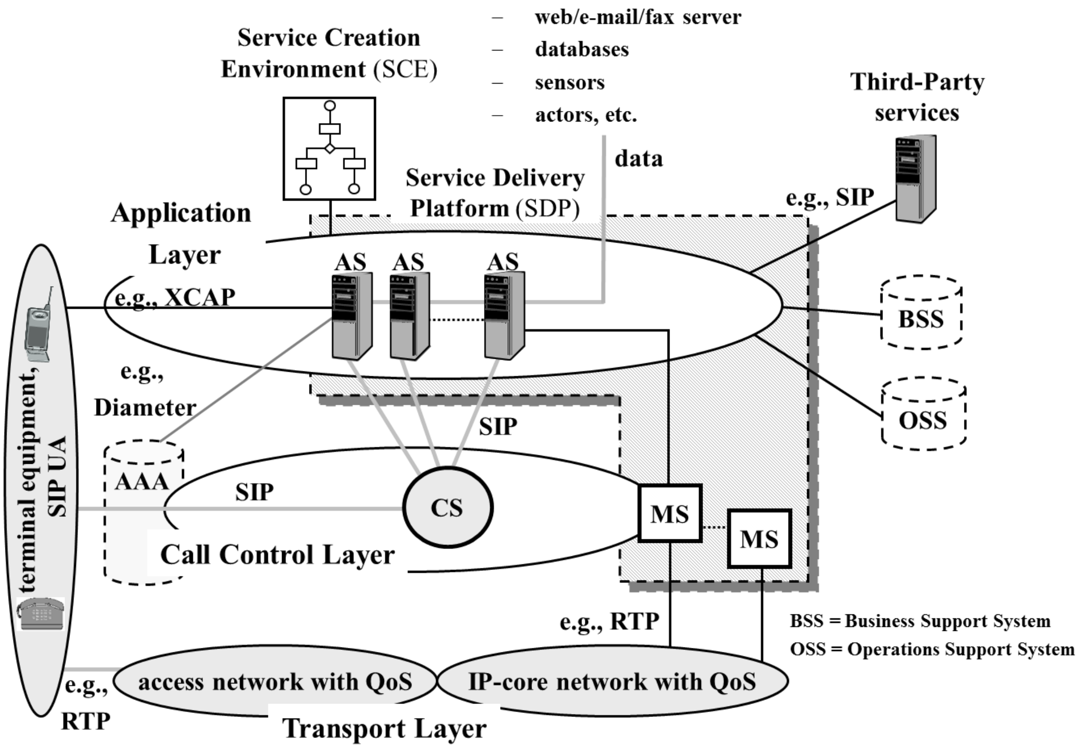


Figure 2.9: Service Delivery Platform in NGN (related to (Trick and Weber, 2009))

A Service Delivery Platform offers the provisioning of own services, the usage of third-party services and the composition of services. It provides interfaces to the Service Creation Environment (SCE), to Authentication, Authorisation, and Accounting (AAA), to the Operation Support System (OSS), and to the Business Support System (BSS) (Lehmann *et al.,* 2007). An AAA system, also called triple-A system, provides protected interfaces to the network elements in the transport network, and controls the access to the network elements. An OSS is a network management system that covers service management and network management. A BSS is a system for the management of business processes (Trick and Weber, 2009).

A Service Delivery Platform can contain multiple application servers and media servers. The Service Delivery Platform is complemented by the Service Creation Environment (SCE). With this SCE, new services can be developed from scratch or from predefined modules. Graphical development tools normally support the service development. As result of the connection between the SCE and the Service Delivery Platform, a direct provisioning of value-added services is possible. (Trick and Weber, 2009)

An overview of the Service Delivery Platform architecture according to Moriana Group (Moriana, 2004) (Moriana, 2013) is shown in Figure 2.10.

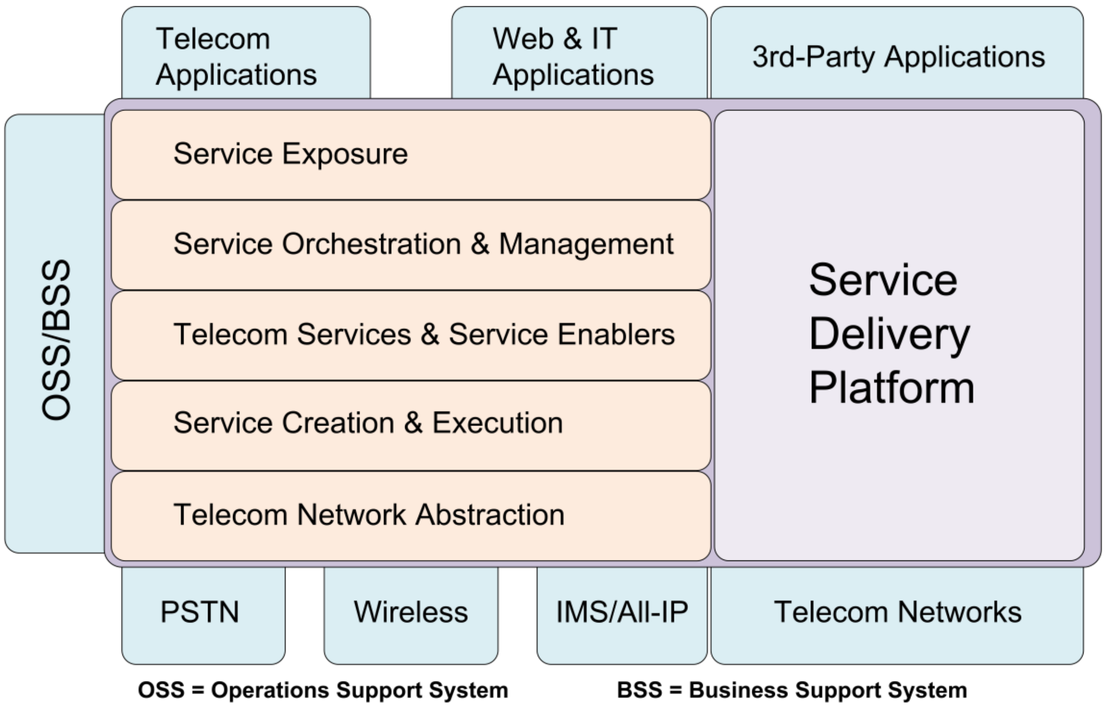

Telecom Applications
Web & IT Applications
3rd-Party Applications
OSS/BSS
Service Exposure
Service Orchestration & Management
Telecom Services & Service Enablers
Service Creation & Execution
Telecom Network Abstraction
Service Delivery Platform
PSTN
Wireless
IMS/All-IP
Telecom Networks
OSS = Operations Support System
BSS = Business Support System

Figure 2.10: SDP architecture (related to (Moriana, 2004), (Moriana, 2013))

The Service Delivery Platform provides interfaces to BSS and OSS; it spans over different networks and provides web, IT, and telecom applications. The Service Delivery Platform architecture consists of five layers: Service Exposure Layer (SEL), Service Orchestration and Management Layer, Telecom Services & Telecom

Enablers Layer, Service Creation and Execution Layer, and Telecom Network Abstraction Layer (NAL).

To make use of the services the Service Exposure Layer (SEL) opens the access for the services to third-party service providers and other companies. For the exposure of the services, secure and standardised interfaces are defined. An abstraction of the interfaces hides the complexity for the developers, so they do not need detailed knowledge for telecommunications. (Mulvenna *et al.*, 2008)

The Service Orchestration and Management Layer can be used for an integration of OSS, BSS, and legacy systems, it follows the principles of SOA (Lu *et al.,* 2008).

The Telecom Services and Service Enablers Layer provide deployable telecom services. To process the services, the corresponding enablers are offered. These enablers are abstract interfaces, which allow services to make use of telecommunication resources (Lu *et al.,* 2008).

The Service Creation & Execution Layer offers an environment to develop, change, configure, deploy, execute, activate, and deactivate services. It supports the creation of new services. Services can also be developed out of a set of predefined and already existing services (Lehmann *et al.,* 2007). This layer can be built around a Java EE (Java Platform, Enterprise Edition), .NET or Telecom Application Server. Services can consist of other services implemented by using different technologies like Java 2 Platform, Standard Edition (J2SE), Java 2 Platform, Enterprise Edition (J2EE), Java Servlets, XML (Extensible Mark-up Language) (W3C, 2008), etc. (Mulvenna *et al.,* 2008)

The Telecom Network Abstraction Layer (NAL) offers standardised abstract interfaces to use the service capabilities of the networks. This layer hides the underlying complexity of networks. (Lu *et al.*, 2008)

2.5 Criteria of Service Creation, Service Execution, and Service Provisioning Technologies for Value-Added Telecommunication Services

In this section the relevant investigation criteria for service execution, provisioning, and creation technologies are defined. Most of the criteria were identified in the study (Lehmann *et al.,* 2008a). The first set of criteria is relevant for service creation technologies. The solutions for service creation in chapter 3 are evaluated regarding these criteria. The second set of criteria is relevant for the service execution and provisioning technologies. The solutions for service execution and provisioning in chapter 4 are evaluated regarding these criteria. From the results of the evaluation, the technologies for service description, creation, and provisioning are derived. The existing technologies may not completely fulfil all of these criteria at once. Some extensions or modifications of the existing technologies may be necessary to fulfil the criteria completely. Moreover, it may be necessary to combine elements of different existing technologies to fulfil the requirements.

The following investigation criteria are relevant for service creation:

- Abstraction from underlying protocols: the developer should concentrate on the description of the service logic. Detailed knowledge of the protocols shall not be required.
- Ability to define a broad range of value-added services: The SCE should not be restricted to one service domain only; for example, call processing in telecommunications, the definition of services from multiple domains, and the support of multiple protocols should be possible. Furthermore, the granularity of the service should be adaptable. Fine-grained service elements should allow a high flexibility and possibilities to modify the service in detail. Coarse-grained service elements should allow an easy and fast service development. The developer can define services by using only a few coarse-grained elements. When it is necessary to define these services in detail, the developer can describe them with fine-grained elements. The services should be abstract from the underlying protocols.
- GUI: a graphical development tool is required for the service description to support an easy and fast service development. It must be able to describe the service logic and the functionality of value-added services.

These criteria will be used to evaluate the service creation technologies in chapter 3.

The following criteria are relevant for service execution and provisioning:

- Supported protocols: for the development of value-added services, a service execution environment is required that supports multiple protocols. It should also be possible to add new protocols to the framework.

- Performance: performance means the execution speed of a typical service (Kuthan, 2000) (Van Den Bossche *et al.,* 2006). In the case of telecommunication services, the Service Execution Environment (SEE) should ensure low latency and high throughput (Maretzke, 2005). The developed framework should be comparable to a typical service execution environment for telecommunication services, e.g., the JSLEE implementation mobicents. Typical values for the latency in telecommunications are in the millisecond range. Therefore, target values within the millisecond range would be a great result.
- The Service Execution Environment shall offer great number of service possibilities: service possibilities indicate how many services can be defined, based on the available functionalities (Kuthan, 2000). It should be possible to add new functionalities to the Service Execution Environment.
- Composition capability/reusability: a composition of existing services or service components for creating new services is desirable in order to prevent that new services must always be developed from scratch. This capability should allow defining service building blocks that can be used in the service creation environment (SCE) described in section 6.2.

These criteria will be used to evaluate the service execution and provisioning technologies in chapter 4.

2.6 Conclusion

This chapter introduces the required basics for service provisioning using NGN. It discussed the concept of network-based service provisioning in NGN, IMS, and the Internet. In the first section (refer to section 2.1), the term “service” was defined for the IT domain and for the telecommunications domain. Furthermore, regarding the telecommunications domain the classification of services were outlined.

The second section (refer to section 2.2) focused on network-based service provisioning. In this section, the architectures in which these services operate were analysed. The characteristics of the Internet and the characteristics of NGN and IMS were discussed, and the support of services in these networks is described.

An overview of the network elements that are required for value-added services in NGN was given in section 2.3. There, the required logical NGN network elements SIP AS (Application Server) and the MS (Media Server) were presented in detail. Section 2.4 described the Service Delivery Platforms for the provisioning of value-added services in next generation networks and explained the general architecture. The criteria for the service creation, provisioning, and execution technologies were defined in section 2.5.

This chapter offered a short overview of the infrastructure addressed in the thesis. The next step is the analysis of currently existing technologies and of related research projects in the sector of service creation, service provisioning, and service execution. The technologies have to be evaluated regarding the defined criteria in section 2.5.

The actual state-of-the art solutions for service creation are analysed in chapter 3. Then, the current solutions for service execution and service provisioning are analysed in chapter 4. From the results of chapters 3 and 4, requirements are defined in section 5.1 that have to be fulfilled by the desired novel framework, which is proposed in this thesis (refer to section 6).

3 Current Solutions for Service Creation

This chapter investigates current solutions for service creation and related research projects. The technologies are evaluated using the objectives and criteria introduced in chapter 2. The evaluation criteria for service creation listed above are a user-friendly development with a GUI, an abstraction from underlying protocols, and the possibility to define a broad range of value-added services.

The evaluated current technologies are technologies that are typically used for service creation in the telecommunication sector where they usually define call-oriented actions and in the IT sector where they are used to describe business processes. In the following section these technologies, are analysed as to whether they are appropriate for creating value-added telecommunication services or not, regardless of the sectors where they are typically applied.

It starts with the relevant current technologies, Call Processing Language (CPL) in section 3.1, Language for End System Services (LESS) in section 3.2, Voice Extensible Mark-up Language (Voice XML) in section 3.3, Call-Control eXtensible Mark-up Language in section 3.4, Service Creation Mark-up Language (SCML) in section 3.5, Web Service Business Process Execution Language (WS-BPEL) in section 3.6, Business Process Model and Notation Language (BPMN) in section 3.7, and Telecommunication Modelling Language (TelcoML) in section 3.8.

In section 3.9, related research projects are analysed, and the problems of the regarded approaches are explained.

3.1 Call Processing Language (CPL)

CPL (IETF RFC 3880, 2004) is an XML-based (Extensible Mark-up Language) (W3C, 2008) language (Rosenberg *et al.,* 1999). Users can write a script or use a graphical design tool to develop a service. To make the service available, the user has to send it to his service provider. CPL was developed for the description of multimedia services (IETF RFC 3880, 2004) and is characterised by the independence of the operating system and the signalling protocol. Through the restricted instruction set, only defined actions can be executed. Thereby, CPL provides a high level of security. Proprietary extensions of the instruction set are possible but can affect security (Trick and Weber, 2009). Through the support of graphical editors like the CPL Editor (Figure 3.1) (Becker, 2015), end users are able to develop their own CPL services, which may be a significant advantage.

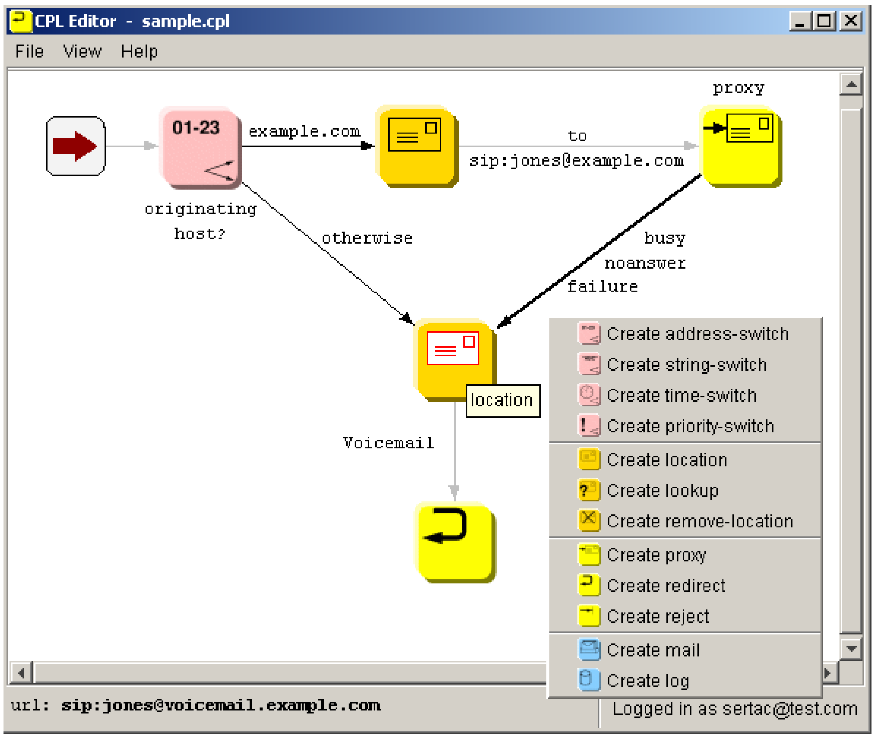


Figure 3.1: CPL Editor (Becker, 2015)

The properties of CPL are discussed in more detail in the following example. A graphical representation of the example service (IETF RFC 3880, 2004) is presented in Figure 3.2. The participant “jones” is reachable in the domain “example.com”. If a call for “jones” is received from the same domain “example.com”, the call is forwarded to his SIP URI (Uniform Resource Identifier) (sip:jones@example.com). In this case, after a proxy timeout of 10 seconds, the call is forwarded to the mailbox, if participant “jones” himself is on the phone (busy), does not accept the call within the 10 seconds, or an error has occurred. If the call is from a different domain, the call is directly forwarded to the mailbox (sip:jones@voicemail.example.com).

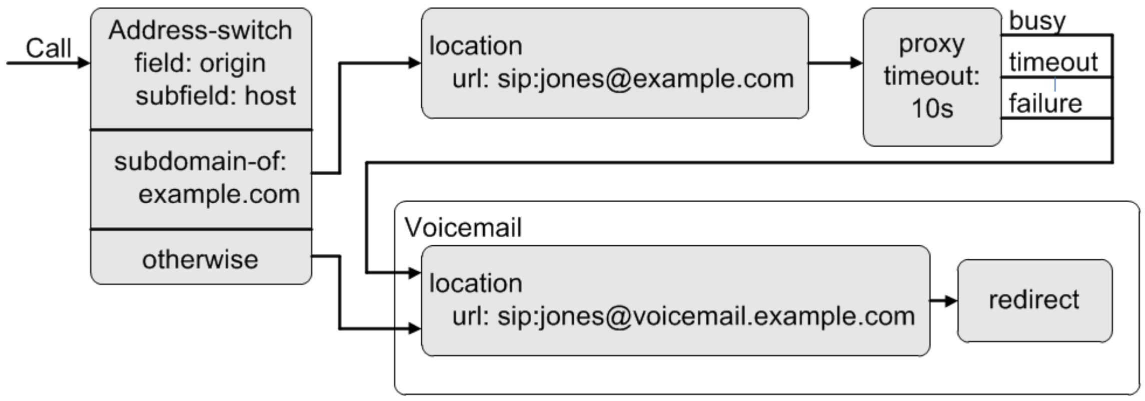


Figure 3.2: Graphical representation of a CPL script (IETF RFC 3880, 2004)

The graphical representation of the CPL script shown in Figure 3.2 can be created with the graphical editor. Figure 3.3 shows the resulting CPL script. This script can be stored on an application server. When a call for “jones” is received, the application server parses the script with the help of a CPL parser and processes it to fulfil the service logic.

CPL allows creating new services very easy and fast, only little telecommunication expertise is required. Unfortunately, CPL has a limited instruction set, only actions which are defined there can be executed.

The extension of the instruction set is possible, but this will result in proprietary solutions and security issues. The language is not Turing-complete, and loops or recursions are not supported (IETF RFC 3880, 2004). Therefore, CPL cannot be used for describing more complex value-added telecommunication services but only simple call processing services like “Call Redirect”, “Call Forward”, “Call Screening”, “Location Filtering”, or “Conditional Routing”. In addition, the development of more complex value-added services inheriting multiple protocols is not supported. Since the protocol support of CPL and the possible services are limited, CPL is not chosen as service description language of this thesis.

```xml
<?xml version="1.0" encoding="UTF-8"?>
  <cpl xmlns="urn:ietf:params:xml:ns:cpl"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="urn:ietf:params:xml:ns:cpl cpl.xsd ">
    <subaction id="voicemail">
      <location url="sip:jones@voicemail.example.com">
        <redirect />
      </location>
    </subaction>
    <incoming>
      <address-switch field="origin" subfield="host">
        <address subdomain-of="example.com">
          <location url="sip:jones@example.com">
            <proxy timeout="10">
              <busy> <sub ref="voicemail" /> </busy>
              <noanswer> <sub ref="voicemail" /> </noanswer>
              <failure> <sub ref="voicemail" /> </failure>
            </proxy>
          </location>
        </address>
        <otherwise>
          <sub ref="voicemail" />
        </otherwise>
      </address-switch>
    </incoming>
  </cpl>
```

Figure 3.3: XML document of a CPL script (IETF RFC 3880, 2004)

3.2 Language for End System Services (LESS)

The Language for End System Services (LESS) (IETF, 2005) (Wu and Schulzrinne, 2003) is an XML-based scripting language. It is an extension of the Call Processing Language (CLP) (IETF RFC 3880, 2004), (Wu and Schulzrinne, 2007) and uses a tree-like structure to describe telecommunication services. It is easy to understand even without programming expertise, offers safety, simplicity, and extensibility. It includes commands and events that provide direct interaction and control of media applications and other end system applications. This language is targeted for end users and for client side services at the telecommunication network end points. It offers a collection of rules to describe the service.

The majority of service languages are designed for network services that run on application servers, and not for user-based end system services. When developing services for end systems, it has to be considered that the call model for end system services and network services is different. Models for a two-party call are shown in Figure 3.4. On the left side, the model for a network service is depicted and on the right side the model for an end system. In the network service model, the service establishes the connection between the communication parties. In the end system service model, the service instructs the local application to send media to and receive media from remote addresses. The different call models have different states, events, and actions. Scripts for network services will be not suitable for end systems, and scripts for end systems will also not be suitable for the application server in the network. (Wu and Schulzrinne, 2003)

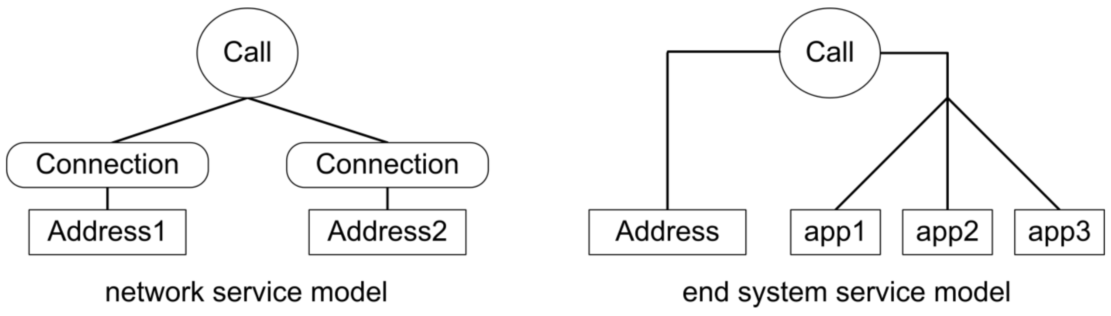


Figure 3.4: Call models of network services and end system services (Wu and Schulzrinne, 2003)

LESS service scripts can be created with a text editor. Another option for the service creation is the development of service templates. LESS templates are written in LESS. The templates use conventions “placeholder” for configurable values. These values can be changed by the service user, e.g. with the help of graphical tools (Wu and Schulzrinne, 2003).

LESS is only defined for the creation of end system services and not for value-added services that run on application servers. Because of this restriction, the possible services are limited and therefore, LESS is not suitable as service creation language in this research project.

3.3 Voice XML (Voice Extensible Mark-up Language)

The Voice Extensible Mark-up Language (VoiceXML) or (VXML) (W3C, 2007a) is a high-level XML-based language for rapid development of voice applications. It allows the user to interact with a voice browser and navigate through voice menus. VoiceXML allows the description of applications that support synthesised speech, playback of digitised audio files and streams, recognition of spoken words and sentences, the recognition of DTMF key input, recording of spoken input, audio dialogue control, and some basic telephony features like call transfer and disconnect. The goal of VoiceXML is to bring the power of web development and content delivery to the voice response applications. The service developer should not need to take care about low-level programming and resource management.

The architecture of VoiceXML is shown in Figure 3.5. The voice applications are executed on an implementation platform (VoiceXML gateway) that implements the required VoiceXML language functionality. A document server (e.g., web server) produces and provides the voice dialogues, handles the service logic, and performs database and legacy operations (W3C, 2007a). The VoiceXML interpreter is responsible for interpreting the VoiceXML scripts. It is contained in the VoiceXML interpreter context, which provides the supported functions that are required by the interpreter.

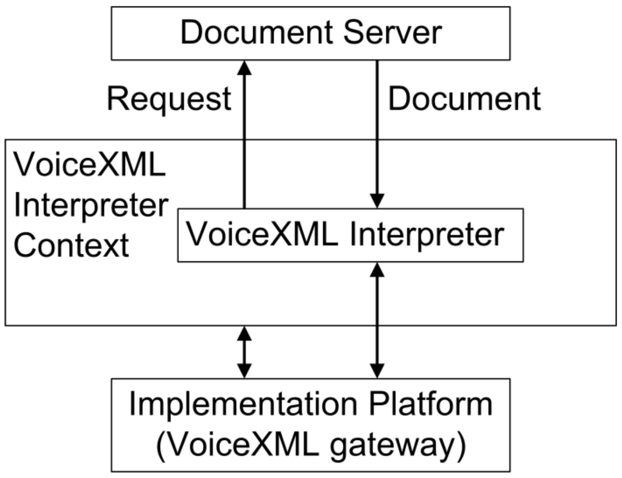

Document Server
Request
Document
VoiceXML Interpreter Context
VoiceXML Interpreter
Implementation Platform (VoiceXML gateway)

Figure 3.5: VoiceXML architecture (related to (W3C, 2004b))

Figure 3.6 presents a typical example of a VoiceXML file that represents a dialogue of a voice application. The application will start with reading out the first <prompt> “Please make your choice:”, which will ask the user for a language. The user can choose among three languages. If the user does not choose a language, the application will ask again (“Please choose:”).

```
<?xml version="1.0"?>
<vxml version="2.0">
  <menu>
    <prompt>
      Please make your choice: <enumerate />
    </prompt>
    <choice next="http://german.spk.com/start.vxml">
      german
    </choice>
    <choice next="http://chinese.spk.com/start.vxml">
      chinese
    </choice>
    <choice next="http://spanish.spk.com/start.vxml">
      spanish
    </choice>
    <noinput>Please choose: <enumerate /></noinput>
  </menu>
</vxml>
```

Figure 3.6: XML document of a VoiceXML dialogue

VoiceXML is a good choice for developing voice applications including user interaction, e.g., voice dialogues. Graphical development tools are available for supporting the service development process, e.g., the IBM WebSphere Voice Toolkit (IBM WebSphere, 2015) (Figure 3.7) which supports CCXML (refer to section 3.4) and VoiceXML. Furthermore, the UML Profile and Metamodel for Voice-based Applications (VOICP) (OMG, 2008) can be used to develop applications in a graphical way by using UML tools.

The language VoiceXML is very limited and does not possess the expressiveness to describe value-added services beyond the scope of voice applications. It is not designed to support advanced call control applications. The restrictions of VoiceXML lead to the conclusion that it is not a suitable solution for this research project.

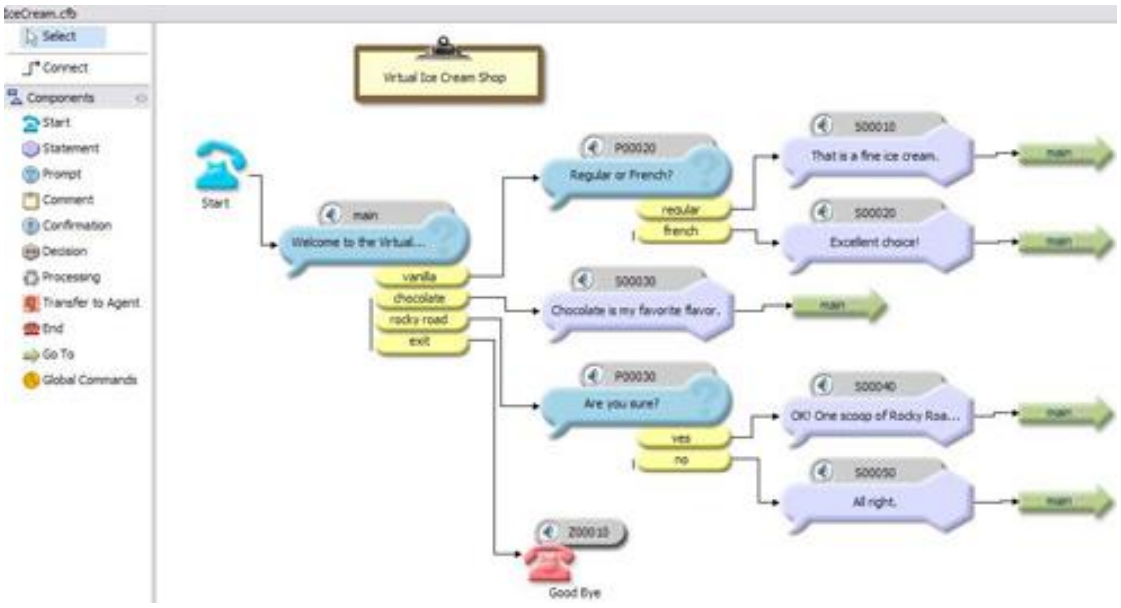


Figure 3.7: IBM WebSphere Communication Flow Builder (IBM WebSphere, 2015) (First published by IBM developerWorks at http://www.ibm.com/developerWorks/websphere/downloads/voicetoolkit.html)

3.4 CCXML (Call-Control eXtensible Mark-up Language)

The Call-Control eXtensible Mark-up Language (CCXML) (W3C, 2011) is an XML-based language that supports call set-up, call monitoring, and call tear-down. It was developed because of the call control limitations of VoiceXML (refer to section 3.3), but it can also be used in combination with other dialogue systems. CCXML supports some more advanced features like multi-party conferencing, conference control, and so on. CCXML is a high-level language for call control on top of telephony platforms.

Figure 3.8 shows an example of a telephony architecture implementation. This architecture consists of four elements: a caller, a dialogue server, a conference server, and the CCXML implementation. The caller is connected via the telephone network. The dialogue server can, e.g., be a VoiceXML implementation. The conference server is used to mix the media streams, and the CCXML implementation manages the connections between the caller and the dialogue server. A telephony web application can also be integrated together with the voice web application. The implementation of the telephony control interface and the dialogue control interface can be an API or a protocol. (W3C, 2011)

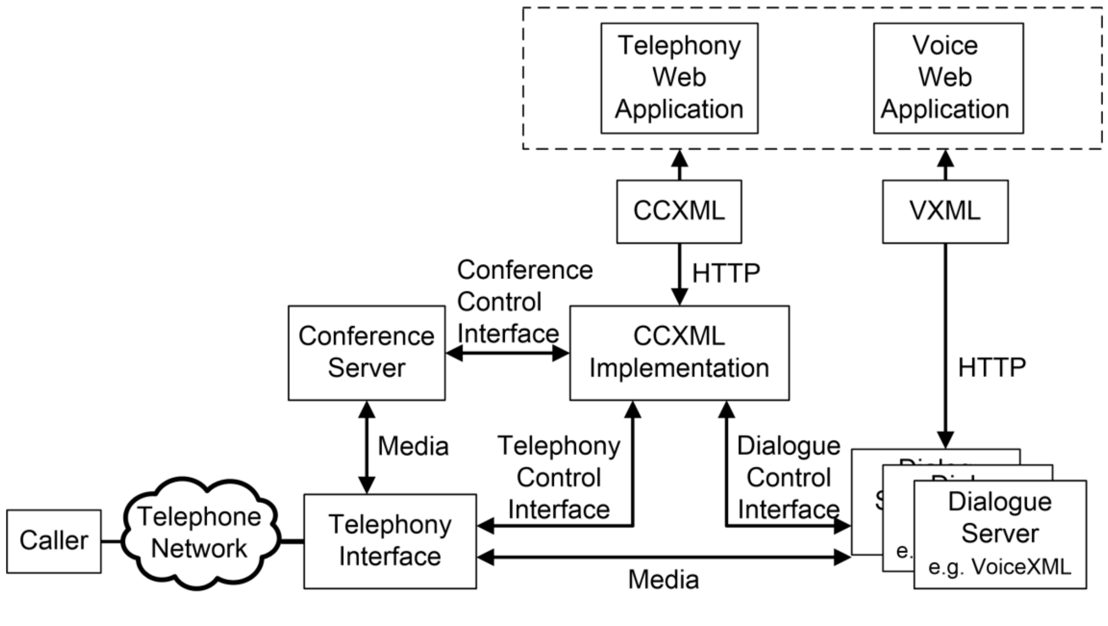

Telephony Web Application
Voice Web Application
CCXML
VXML
Conference Control Interface
HTTP
Conference Server
CCXML Implementation
HTTP
Media
Telephony Control Interface
Dialogue Control Interface
Caller
Telephone Network
Telephony Interface
Media
Dialogue Server e.g. VoiceXML

Figure 3.8: CCXML system architecture (related to (W3C, 2011))

CCXML supports multi-party conferencing, more advanced conferences than VoiceXML, audio control, voice applications that support its own dedicated VoiceXML interpreter for each active line (not possible in this way in VoiceXML), multiple-call handling and control, handling of asynchronous external events, and interaction with outside call center platforms via events.

The CCXML service creation can be done with text, XML und GUI editors. One example of a graphical development tool is the IBM WebSphere Voice Toolkit (IBM WebSphere, 2015) which supports CCXML and VoiceXML (Figure 3.7).

However, the creation of more complex value-added services with CCXML is not possible. This language offers more advanced application features related to call control than VoiceXML, but the ability to describe general value-added services is rather limited. CCXML is therefore not a suitable service description language for this work.

3.5 SCML (Service Creation Mark-up Language)

The Service Creation Mark-up Language (SCML) (Bakker, 2002), (IETF, 2001c) is a XML-based scripting language based on JAIN JCC (Java Call Control) API (JSR 21, 2002). SCML is a protocol-independent high-level interface abstraction API for describing services in NGN. It hides the complexity of the underlying network and is easy to use, similar like CPL, but more flexible (Licciardi, 2003).

An overview of the SCML architecture is given in Figure 3.9 (IETF, 2001c). The architecture consists of the four elements Capability Server (e.g., SCF or softswitch), Client, Gateway, and SCML server. In the communication between the elements, the three interfaces A, B, and C are involved.

The Capability Server executes the SCML service logic commands issued from the SCML Server. It communicates via the Client with the elements of the IP domain, and interacts with the underlying transport network elements.

The Client receives requests from the Capability Server, sends responses to the Capability Server, and forwards requests from the Gateway to the Capability server. The communication between Capability Server (e.g., SCF) and client is done via interface C.

The Gateway communicates on one side with the Client via the interface B and, on the other side, with the SCML Server via interface A. The subscriber receives events through the interface A, and the gateway receives the script's disposition of the call and initiates the services. The Gateway can communicate with the SCML Server, or it may act as a virtual server, terminating the requests without sending them to the Server.

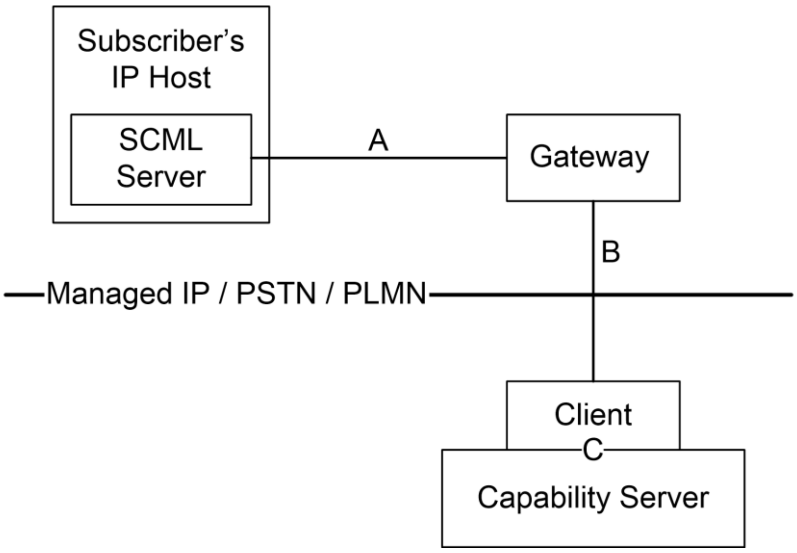


Figure 3.9: SCML architecture (related to (IETF, 2001c))

The SCML Server executes the SCML scripts. It issues requests to be executed on the Capability Server, and terminates requests or events from the Capability Server.

SCML is like CPL a scripting language. It is more flexible than CPL, but it is also not capable to describe multimedia value-added services that span across multiple protocols. This language is, like CPL, also not suitable for service description in this research project.

3.6 WS-BPEL

The Web Services Business Process Execution Language (WS-BPEL) is an XML-based language to specify business processes. The activities of the business process are implemented as web services. In 2002, Microsoft, IBM, and BEA specified BPEL and named it BPEL4WS 1.0 (Curbera *et al.,* 2002). In 2004, the version 2.0, with the name WS-BPEL 2.0, was defined by OASIS (Organization for the Advancement of Information Standards) (OASIS, 2007).

BPEL was originally developed to orchestrate web services, i.e. to compose new web services from multiple distributed web services. Thus, BPEL processes can call other web services and, at the same time, be called by them (Figure 3.10).

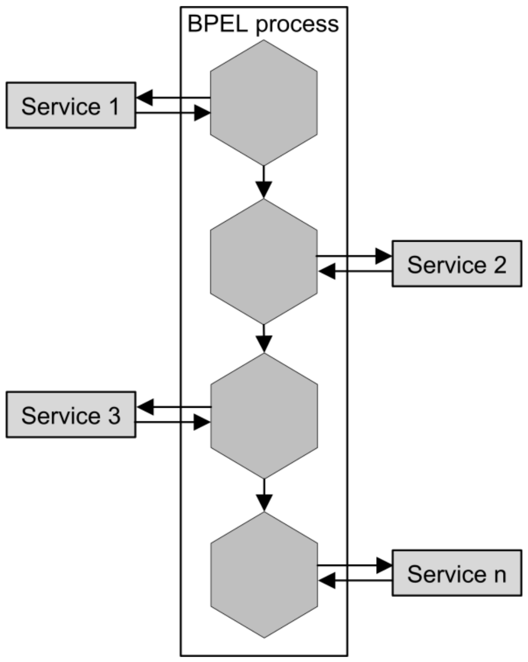


Figure 3.10: BPEL process in cooperation with other services

These services are called “Partners” in the BPEL process. The interfaces of the BPEL processes are described with Web Services Description Language (WSDL). The SOAP protocol is used for the message exchange (Figure 3.11).

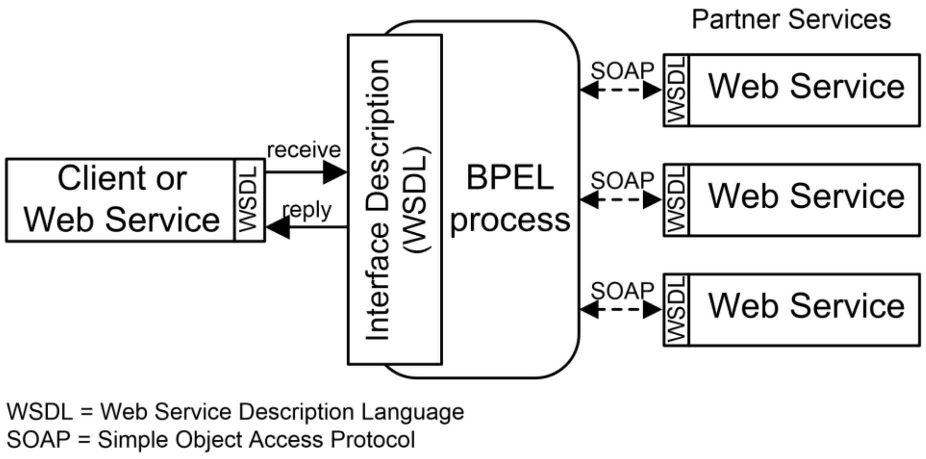

Partner Services
Client or Web Service
WSDL
receive
reply
Interface Description (WSDL)
BPEL process
SOAP
Web Service
WSDL = Web Service Description Language
SOAP = Simple Object Access Protocol

Figure 3.11: BPEL process with interface and message protocol

In Figure 3.12, a graphical representation of a simple BPEL process is illustrated. This example was created with the BPEL plugin of the Oracle IDE (Integrated Development Environment) JDeveloper (JDeveloper, 2014). This process realises a database query. The Receive activity “receiveInput” gets a request from the client and formulates the database query in the first assign element “Assign_SubscriberID-to-DB”. With the Invoke activity “Invoke_DB”, the query is sent to the database “db1”. With the result of the database query, the process is preparing the response for the client in the second assign element “Assign_DBSubscribername-to-Output”, and sends it back with the Reply activity “replyOutput”. If any errors occur during the execution of the BPEL process, the exception handling will be activated by the Throw activity. There, the Assign activity “Assign-ErrorString-to-Output” creates an error message, which is sent back to the client.

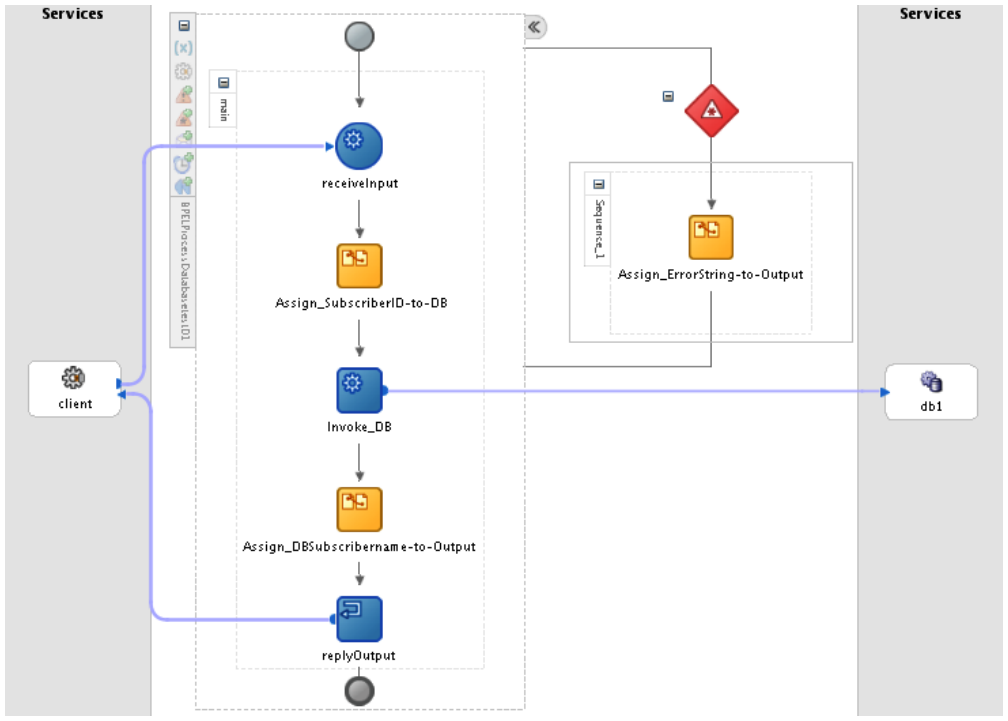

Services
BPELProcessDatabasetest01
main
receiveInput
Assign_SubscriberID-to-DB
Invoke_DB
Assign_DBSubscribername-to-Output
replyOutput
Sequence_1
Assign_ErrorString-to-Output
client
db1
Services

Figure 3.12: A simple BPEL process (created with Oracle IDE JDeveloper (JDeveloper, 2014)

To sum it up, the GUI support of BPEL offers a user-friendly possibility to define business processes. BPEL is open for third-party development, and with the help of the partner links BPEL can interact with other web services.

BPEL is normally used in combination with web services, but as described in chapter 4, web services are not the solution that this thesis is aiming at. Instead of web services, JAIN SLEE was selected as SEE. Therefore, the service description language must be able to describe JAIN SLEE-based value-added services. Section 5.2 introduces a novel approach that offers the possibility to describe the service logic and the functionality of a value-added JAIN SLEE service with BPEL.

3.7 BPMN (Business Process Model and Notation)

The Business Process Model and Notation (BPMN) (OMG, 2011b) language was developed as a graphical representation for specifying business processes. The goal of BPMN is to provide a notation that is usable for technical users and business users as well.

The example process in Figure 3.13 illustrates the core concepts of BPMN (OMG, 2010). The example process is not an executable process model. It focuses on organisational aspects of a business process. This process describes the steps a hardware retailer has to fulfil before the ordered goods can actually be shipped to the customer. This example uses one pool (Hardware Retailer) and different lanes for the people involved in this process (Warehouse Worker, Clerk, and Logistics Manager). A pool represents major participants in a process and contains one or more lanes. Lanes organise and categorise activities within a pool according to function or role.

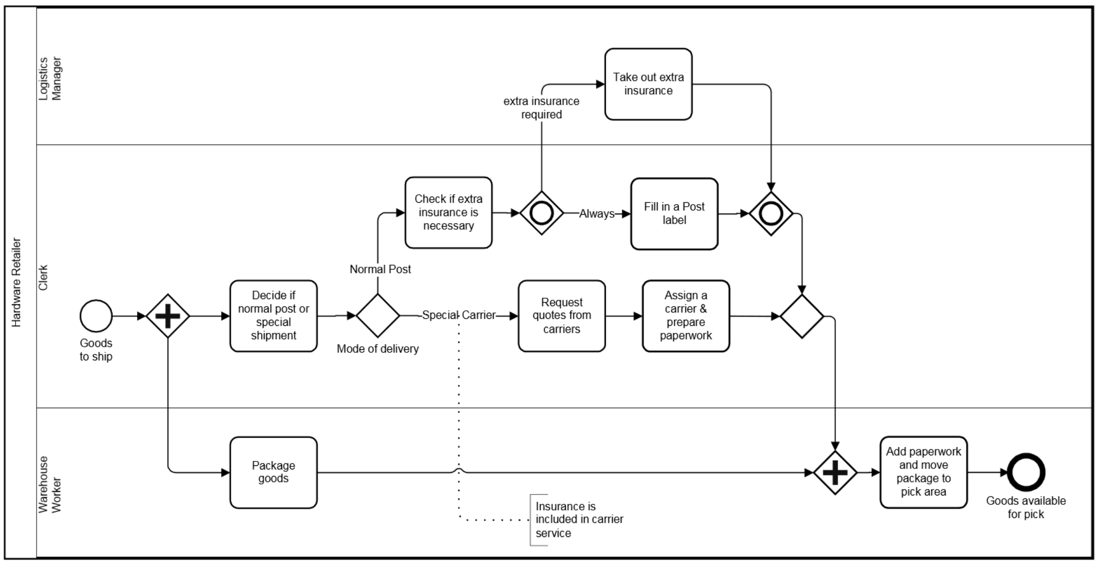

Hardware Retailer
Logistics Manager
Clerk
Warehouse Worker
Goods to ship
Decide if normal post or special shipment
Mode of delivery
Normal Post
Special Carrier
Check if extra insurance is necessary
extra insurance required
Take out extra insurance
Always
Fill in a Post label
Request quotes from carriers
Assign a carrier & prepare paperwork
Package goods
Insurance is included in carrier service
Add paperwork and move package to pick area
Goods available for pick

Figure 3.13: BPMN example process (OMG, 2010)

The process starts with the start event "Goods to ship". The next element in the process is the parallel gateway. This indicates that there are more functions that can be executed in parallel. The Warehouse Worker has to "Package the goods" and the Clerk has to "Decide if normal postal or special shipment".

The next element in the Clerk's lane is the exclusive gateway "Mode of delivery". As a router, the gateway provides alternative paths. Whether the path "Normal Post" or "Special Carrier" is selected, depends on the result of the previous element.

If the path "Special Carrier" is selected, the Clerk has to execute the activities "Request quotes from carriers" and then "Assign a carrier & prepare paperwork". In case that the path "Normal post" is selected, the Clerk has to "Check if extra insurance is necessary". This activity is followed by an inclusive gateway with two outgoing paths, the "Always" and the "extra insurance required" path. The "Always" path is taken in any case, independent of the Clerk's decision whether an extra insurance is required or not. Therefore, the Clerk executes the activity "Fill in Post label". If the Clerk decides that an extra insurance is required, the Logistic Manager can execute the activity "Take out extra insurance" in parallel to the Clerk's activities.

The next element is again an inclusive gateway, which is used for synchronisation. This gateway waits until both parallel tasks of the Logistic Manager and the Clerk have been finished. The exclusive gateway which follows synchronises the "Mode of delivery" gateway. Before the last task "Add paperwork and move package to pick area" can be executed, the parallel gateway ensures that both the Warehouse Worker and the Clerk have finished their work.

In the original specification of BPMN, no interchange format was defined for the BPMN applications. In 2011, the XML format was defined (OMG, 2011b). In this format it was described how BPMN documents have to be exchanged between different applications. The BPMN specification also includes a definition of the mapping from a BPMN model to BPEL. For this reason, BPMN can be used on top of BPEL as a graphical service description tool. Since BPEL can be completely mapped to BPMN and vice versa, both technologies seem to be suitable for this thesis. The BPMN description can then be transformed into BPEL (refer to section 3.6) to receive executable BPEL processes. The problem with this transformation is that the resulting processes can become very complex and are not human readable anymore. Therefore, in this thesis, BPEL is more preferred than BPMN.

However, at the time of the evaluation of the service description solution, the XML-based format definition has not yet been completed (OMG, 2011b). Therefore, BPMN was not chosen as service creation environment.

3.8 TelcoML (Telecommunication Modelling Library)

The Telecommunication Modelling Library (TelcoML) specification (OMG, 2013) defines an UML profile for advanced and integrated telecommunication services. It provides extensions to SOAML (Service-Oriented Architecture Modelling Language) (OMG, 2012b). SOAML is an extension to UML 2 that supports service modelling. In fact, TelcoML provides extensions to SOAML with respect to real-time communication services and many of the existing communication services. The main goal is to provide a common abstraction to the existing communication services standards, so that tools can be built for Communications Service Providers to model services in a consistent manner.

The specification consists of two parts. The first part defines the TelcoML Enabler Library. This is a UML representation of a set of service interfaces. This set contains some typical telecommunication enablers: “Generic Messaging”, “Click To Call”, “Synchronisation”, “Voice recognition and TTS”, and “Privacy”. Thus, relevant enablers that are defined in SOAML (OMG, 2012b) are now formulated as TelcoML API facilities (IBM 2012).

The second part, called “Composition Profile” of the specification, defines a convention to represent service compositions.

A TelcoML example service is shown in Figure 3.14 (OMG, 2013). The name of the service is “Send_by_SMS_the_weather_in_Paris_translated_in_english”. The service is a simple composition of the TelcoML telecom enabler “Messaging” and the two capabilities “MeteoFrance” (weather forecast) and “Translator”.

The service starts with the definition of the composed services “Messaging”, “MeteoFrance”, and “Translator”. Then the sequencing of the service calls is specified. The first method is called “`getWeatherForecast(“Paris”,when)`” from the “MeteoFrance” service. The parameters for this method are the location (Paris) and the time (when). The result of the method is stored in the variable (r1).

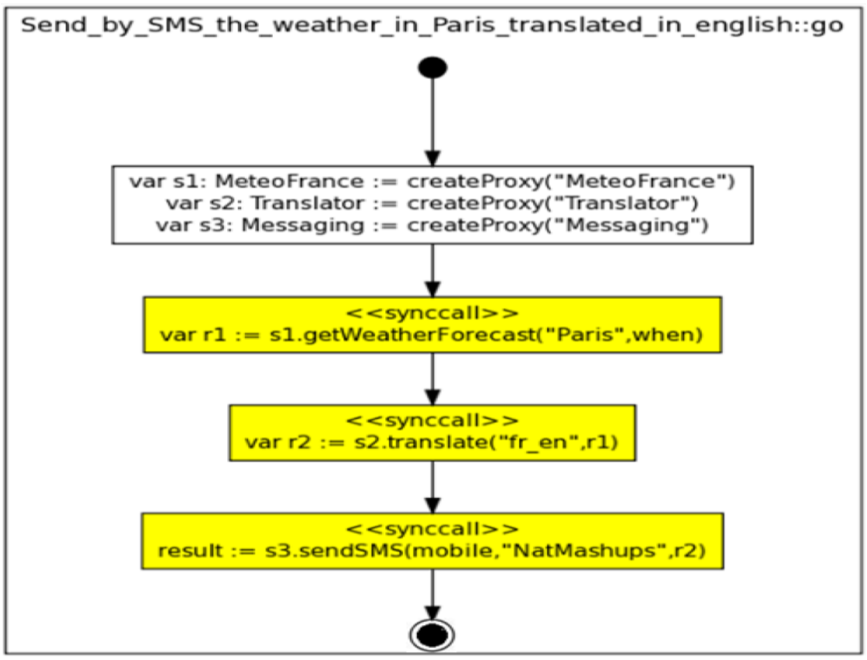

Send_by_SMS_the_weather_in_Paris_translated_in_english::go
var s1: MeteoFrance := createProxy("MeteoFrance")
var s2: Translator := createProxy("Translator")
var s3: Messaging := createProxy("Messaging")
<<synccall>>
var r1 := s1.getWeatherForecast("Paris",when)
<<synccall>>
var r2 := s2.translate("fr_en",r1)
<<synccall>>
result := s3.sendSMS(mobile,"NatMashups",r2)

Figure 3.14: TelcoML SMS example (OMG, 2013)

The next method in the service is the "`translate("fr_en",r1)`" method from the "Translator" service. This method translates the input String into the destination language. The first parameter defines the source and destination language (fr_en), and the second parameter defines the String that should be translated. In the latter case, the weather forecast result (r1) is translated from French into English. The result is stored in the variable (r2).

The last method in Figure 3.15 is "`sendSMS(mobile,"NatMashups",r2)`" of the "messaging" service. This method sends a SMS to the phone number "mobile". The parameters are the recipient's phone number (mobile), the sender ("NatMashups"), and the translated weather forecast (r2).

The service interface (Figure 3.15) requires the two parameters "when" and "mobile". The first parameter "when" defines the time for the weather forecast, and the second parameter "mobile" defines the destination number of the SMS.

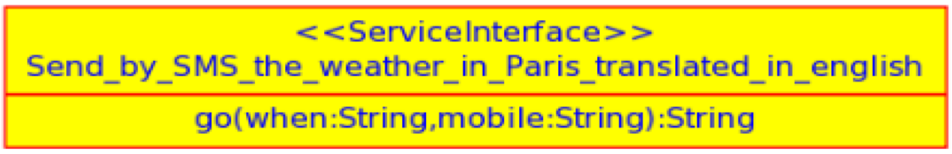

<<ServiceInterface>>
Send_by_SMS_the_weather_in_Paris_translated_in_english
go(when:String,mobile:String):String

Figure 3.15: Interface of the composite service (OMG, 2013)

TelcoML is dedicated to telecommunication services and is protocol-independent. The functionality is limited to the TelcoML enabler library. At time of writing, only five enablers have been standardised. Therefore, the range of possible services is limited. However, through an interworking with SOAML, the range of possible services can be enhanced. TelcoML was not chosen as service description language, because other languages like BPEL and BPMN are more widely spread in the market. For BPEL, more development tools are available, and the GUI is more user-friendly than the TelcoML GUI. Furthermore, TelcoML is a new language, which has not been available at the beginning of this research work. Therefore, TelcoML has not been chosen as service description language.

3.9 Related Research Projects

This section provides an overview of relevant related research project in the area of service creation and service provisioning.

High-Level Service Creation Environment

The first research project (Glitho *et al.,* 2002) and (Glitho *et al.,* 2003) provides an overview of a high-level graphical SCE, consisting of a graphical user interface (GUI) called “SINTEL” (Figure 3.16).

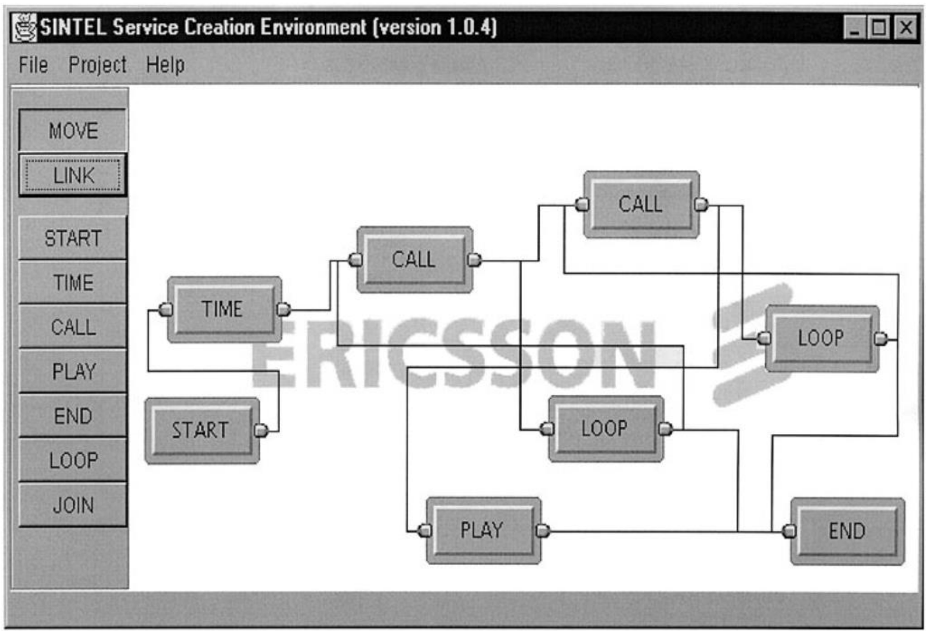


Figure 3.16: Snapshot of high level SCE (Glitho *et al.,* 2003) © 2003 IEEE

A service can be developed out of the eight functions Start, Timer, Call, Loop, Join, Sync, Play, and End. The service logic can be developed with these functions within the GUI. The scope of the experiment is limited to services that originate calls, such as Wake-up Call or Third-Party Call. The services can be executed in a developed SLEE that is based on a Parlay/Session Initiation Protocol (SIP) Gateway implementation. Therefore, the experiment only supports the SIP protocol and only allows basic telephony services. Furthermore, the services have to be compiled. The architecture consists of the SCE, a Common Object Request Broker Architecture (CORBA) (OMG, 2012a) as middleware, a Service Logic Execution Environment (SLEE), a Parlay/SIP gateway, and a SIP network (Figure 3.17).

The Java-based SCE provides the graphical user interface (GUI) with the pre-defined service elements. The service logic can be formulated by placing the service elements on the workspace via drop-down functionality. The services can be defined by connecting the services within the workspace through arrows.

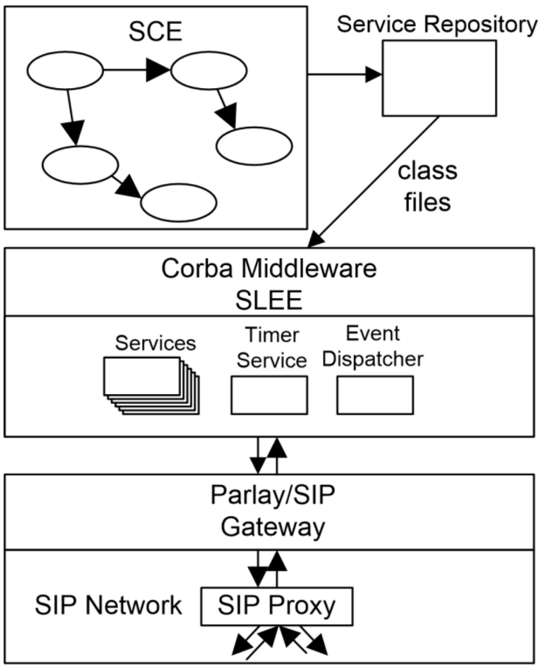


Figure 3.17: High-level service creation environment (Glitho *et al.*, 2002) © 2002 IEEE

The service logic can be developed with the eight functions within the GUI. From the developed service description, the source code of the services will be generated. This source code can be compiled and executed in the SLEE. The SLEE provides a layer that transforms the abstract methods into API-specific interfaces. The CORBA/Internet-Inter-Object Request Broker (ORB) protocol (IIOP) (OMG, 2012a) is used for communicating between the Parlay application interfaces and the service interfaces.

The Parlay/SIP gateway communicates with the SIP network. The services can be executed in a self-developed SLEE (Service Logic Execution Environment) that is based on a Parlay/SIP Gateway implementation. The goal of this project is to find a high-level service description language that is suitable for non-expert service developers.

The SLEE implementation of the Parlay/SIP functionalities support only the SIP protocol, so only services based on SIP can be realised. Another problem is to find out the right level of abstraction. With a higher level of abstraction, the development of services is easier, but the diversity and richness of services is reduced. With a lower abstraction level, only experts can develop the services.

This approach cares about the service creation environment and the service execution. However, in this thesis it is not suitable as service description language, since it only supports a limited functionality. Furthermore, it is not capable of supporting multiple protocols, and it was developed for basic services only. Additionally, the services have to be compiled before execution and cannot be composed from existing components.

SPICE

SPICE (Service Platform for Innovative Communication Environment) (SPICE, 2013) is a project in the field of Service Creation/Execution Environments for mobile services. SPICE was a European IST project and part of the Wireless World Initiative (WWI) that aimed to develop a software architecture that allows a fast and easy creation of new services. The framework of SPICE is independent of network technologies and service providers. The focus of the project is on the creation of services that can be made available across different operator domains and over different countries. Network operators, service providers, and content providers can individually arrange the services. Additionally, the SPICE project integrates existing networks and supports various service platforms. In SPICE, an extendable overlay architecture and framework is proposed to support easy and quick creation, test, and deployment of mobile communication and information services (SPICE NEC, 2013).

The service development is separated into professional service development and end-user service creation (Drögenhorn, 2008). On the professional side, a graphical service description language, named SPATEL, which is based on the UML (OMG, 2011a) notation, is used for the definition of service interfaces and service logic. From this description language, the skeleton of the service can be generated. The UML diagram in Figure 3.18 shows an example SPATEL composite service logic. The state machine describes a translation service. Two translation services are available, one translation service for paying customers and one free translation service.

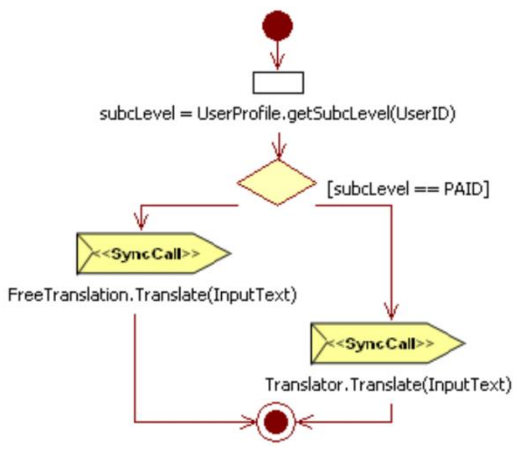

subcLevel = UserProfile.getSubcLevel(UserID)
[subcLevel == PAID]
<<SyncCall>>
FreeTranslation.Translate(InputText)
<<SyncCall>>
Translator.Translate(InputText)

Figure 3.18: SPATEL composite service logic (Drögenhorn, 2008) © 2008 IEEE

On the end user side, services can be developed using a graphical development tool called "End User Studio". The services that are developed by professional developers are available as building blocks in the End User Studio. These building blocks can be composed to services that are more complex. The development tool offers "if then" relations and logical connections like "AND" and "OR" to formulate conditions. Figure 3.19 presents a screenshot of the SPICE End User Studio.

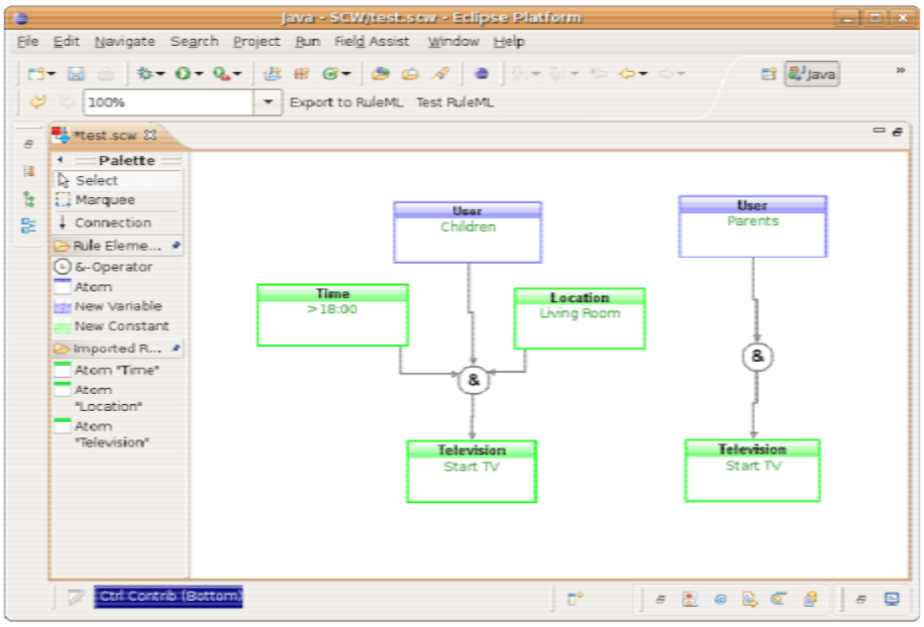


Figure 3.19: SPICE End User Studio (Drögenhorn, 2008) © 2008 IEEE

The SPICE platform requires professional service developers who develop service building blocks. The end user can compose the building blocks to more complex services. The service building blocks among which the end user can choose are therefore coarse-grained.

This limitation of the "End User Studio" hinders the developer to define a broad range of value-added services. Other service description languages like WS-BPEL or BPMN offer more expressivity than the SPICE "End User Studio". The SPATEL service description language was designed for professional developers. It is used for defining the service interfaces and the service logic. The output of SPATEL is the service skeleton. Therefore, SPATEL is ruled out as service description language in this thesis.

A SIP-based Programming Framework for Advanced Telephony Applications

This approach (Jouve *et al.,* 2008) introduces a programming framework that raises the abstraction level for the programming of telephony applications. It provides the programmer with a declarative language called "DiaSpec". With this language, the telephony entities are defined. An entity is characterised by its interaction modes, consisting of the SIP-native interaction modes, namely messages, events and sessions.

The DiaSpec entity declarations are then passed to DiaGen (Figure 3.20). DiaGen is a generator that creates a high-level framework for the Java programming language.

In the first step, the area compiler parses the DiaSpec entity declarations. The consistency is analysed, and the area-specific programming framework is generated. With this framework, the developer can build the service source code. In the last step, the service compiler uses the service code to generate the class files. (Jouve *et al.,* 2008)

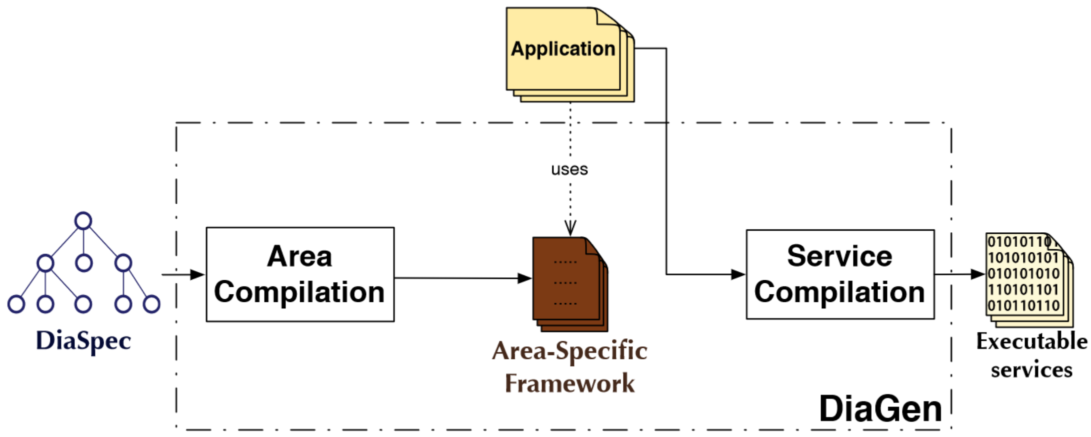


Figure 3.20: DiaGen processing chain (Jouve *et al.,* 2008)

This framework provides service discovery and high-level communication mechanisms. DiaGen generates high-level Java methods that can be used by the programmer to develop telephony applications (Jouve *et al.,* 2008). This approach does not support general value-added services, since it is restricted to the creation of SIP-based telephony applications.

To provide access to non-telephony resources (e.g., database look-up), a command mode that is an RPC (Remote Procedure Call)-like mechanism is offered to invoke operations.

This project offers no graphical service description tool that would allow also non-experts to describe value-added services. Therefore, in this thesis, this approach as service description language is no option.

Session Processing Language (SPL)

SPL (Burgy *et al.,* 2006) is a domain-specific language (DSL) that allows the development of robust telephony services, and offers an abstraction of the underlying protocols and software layers (SPL, 2013). In Figure 3.21, an example counter service is depicted. This service controls a counter of SIP calls. If a call for the service user is forwarded to the secretary, the counter will be increased. When the user registers, the counter is set to zero. When the user unregisters, the counter is logged.

This language is developed around the "SIP Service Logic Execution Environment" (SIP-SLEE) for SIP (Burgy *et al.,* 2006). SIP-SLEE and SPL were implemented during the project. SIP-SLEE provides a high-level design framework for service development.

```
service sec_calls {
  processing {
    local void log (int);

    registration {
      int cnt;

      response outgoing REGISTER() {
        cnt = 0;
        return forward;
      }

      void unregister() {
        log (cnt);
      }

      dialog {
        response incoming INVITE() {
          response r = forward;
          if (r != /SUCCESS) {
            cnt++;
            return forward 'sip:secretary@nist.gov';
          } else
          return r;
        }
       }
     }
   }
 }
```

Figure 3.21: SPL counter service (SPL, 2013)

SPL is specially designed for IP telephony services, but since it only addresses routing logic, it is not sufficient to describe value-added multimedia services.

To enable the description of telephony services also for non-programmers, a graphical service creation and execution environment called “VisuCom” (Figure 3.22) was defined on top of SPL (Latry *et al.,* 2007).

In Figure 3.22, an example service is presented. The service describes a scenario were the user Bob is in a meeting and receives a phone call. In the first step, the caller is checked. If the caller is not Bob´s boss or not a member of the committee the call is rejected, otherwise, the subject of the call is analysed. If the subject contains not the word “important”, the call is rejected; otherwise, the call is redirected to Bob´s current phone. If Bob is not reachable on the current phone, the call is redirected to his cell phone.

VisuCom offers intuitive visual constructs and menus that permit users a quick development of telephony services.

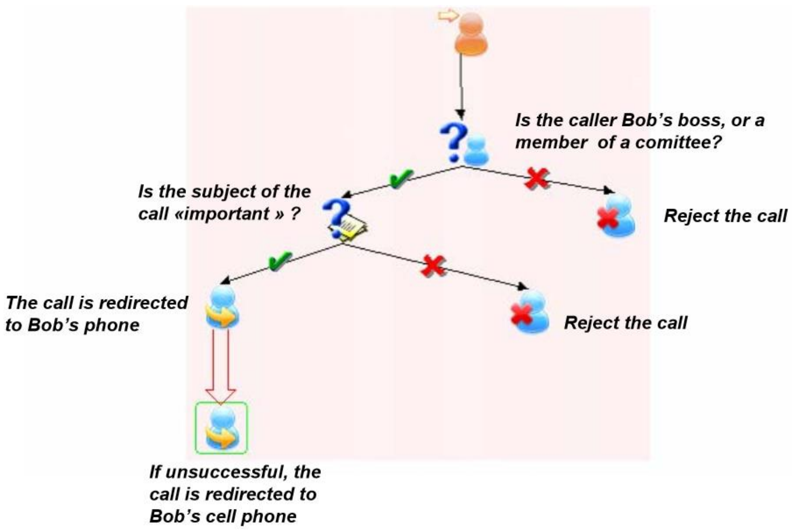

Is the caller Bob's boss, or a member of a comittee?
Is the subject of the call «important » ?
Reject the call
The call is redirected to Bob's phone
Reject the call
If unsuccessful, the call is redirected to Bob's cell phone

Figure 3.22: VisuCom example call routing (Latry *et al.*, 2007)

Since VisuCom is based on SPL, it is liable to the same restrictions as SPL and, therefore, it is not suited to describe value-added multimedia services.

OPUCE

The European IST project OPUCE (Open Platform User-centric Service Creation and Execution) "bridge the advances in networking, communication, and information technology services towards a unique service environment where personalized services will be dynamically created and provisioned by the end-user itself regardless of ambiance and location" (istworld, 2014). The project offers an "open service infrastructure" which enables service providers to create and deploy services. These services are called "base services". Multiple devices connected via various networks can access these services. OPUCE's goal is to become the next generation telecommunications delivery platform that enables the convergence of voice, video, and data (OPUCE, 2010).

The base services are provided by the platform owner or by third parties. The base services provide basic functionalities (e.g., sending SMS or IM, placing a phone call, setting up an audio feed, interacting with web forms, etc.) wrapped and exposed as atomic enablers to the OPUCE platform. High-level services can be composed by a set of base services. (Cipolla *et al.,* 2007)

OPUCE allows the users to create their own OPUCE services from the base services. In order to be able to create OPUCE services, a web-based GUI based on mashups and a mobile editor is provided. A mashup is a web application or web page that integrates other, already existing content from other sources. The output of the mobile editor or the web editor is a XML service description (Sienel *et al,* 2009). The OPUCE services orchestrate the base services. A snapshot of the OPUCE web editor is shown in Figure 3.23.

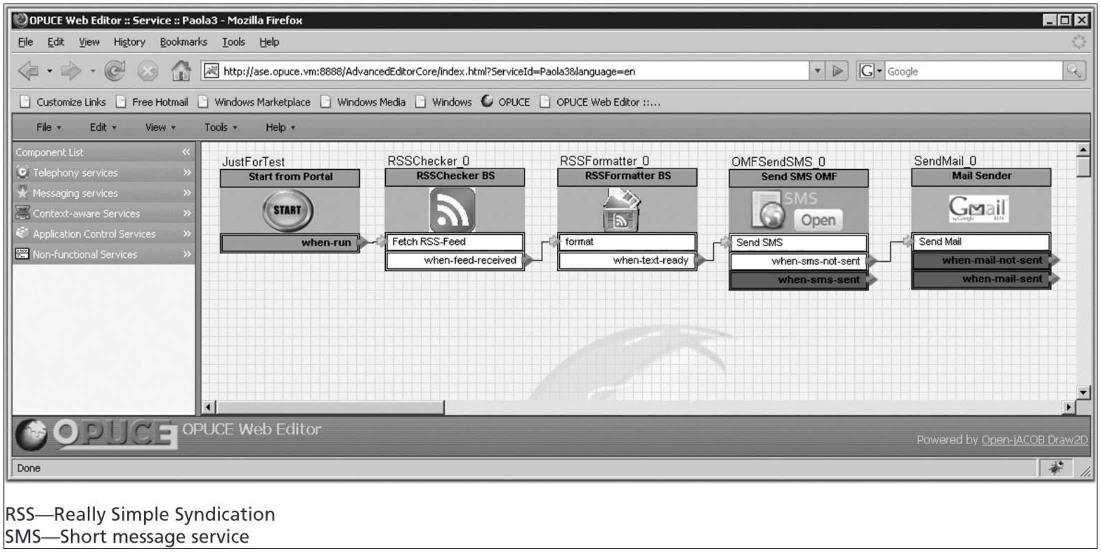


Figure 3.23: OPUCE web editor (Sienel *et al,* 2009), “Reprinted with permission of Alcatel-Lucent USA Inc.”

An example of an OPUCE service composition is shown in Figure 3.24. The three blocks “ReceiveIM”, “TPCC”, and “SendIM” represent base services. Each block offers actions and events that can be used by the service developer to build high-level OPUCE services.

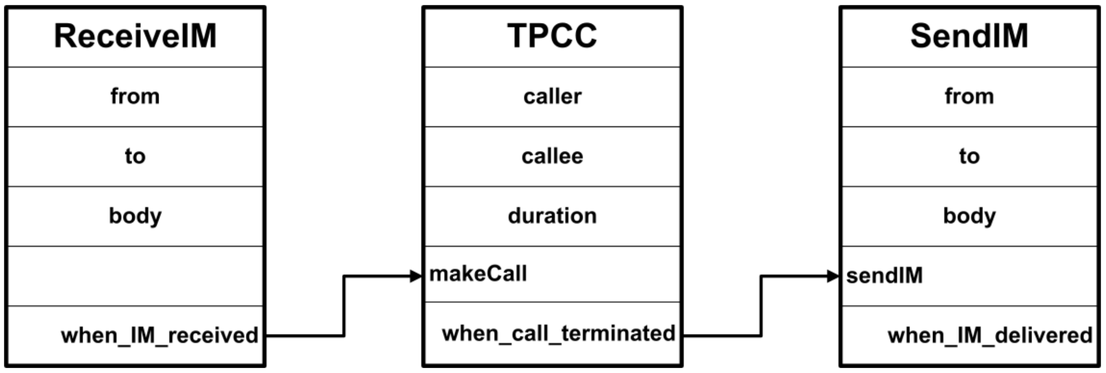


Figure 3.24: Example of an OPUCE Service Composition (related to (Cipolla *et al.,* 2007))

The architecture consists of three big blocks. The first block is the Service Execution Environment (SEE) with the base services and the WSDL interface between the base services and the SCE. The base services can be implemented in any technology, e.g., JAIN SLEE or Java EE. The developed OPUCE service is translated into BPEL and executed on a BPEL engine.

The second block is the Portal. This is the interface between the user and the platform. The platform provides simple tools to create, share, and customise mashups, to discover and subscribe for services, and carry out profile management.

The third block is the OSS (Operation Support System). The OSS bridges the portal and the SEE. It consists of tools for deployment, scheduling, and provisioning, a service repository, security features, and user management.

However, the base services in OPUCE are already complete services. The base services themselves are fully runnable services and not building blocks, which are only part of a service. This leads to coarse-grained high-level services. It is not possible to create the base services with the OPUCE SCE. The base services have to be built manually.

OPUCE uses a BPEL engine as service execution environment. The attempt to define fine-grained low-level building blocks and orchestrate them with a BPEL engine will lead to performance issues (refer to section 3.6). Therefore, orchestration of web services with a BPEL engine is only suitable for coarse-grained services that will limit the possibilities of service creation. Furthermore, the mashups are specially developed to work together with OPUCE so that only high-level services can be composed from the base services. Altogether, this approach is not a suitable solution for this PhD thesis.

TeamCom

The TeamCom project (TeamCom, 2010), (Lehmann *et al.,* 2009) was established to analyse IMS- or P2P-based Service Provisioning and Creation for Customer Tailored Communication Processes. The aim of this approach was to obtain, for the very first time, an easy-to-use, cost-efficient, and fast provision of services, especially for B2B communication: evaluation of IMS and/or P2P communication, estimation of reusable communication elements, and realisation of communication elements via a Service Creation Environment. (TeamCom, 2010)

The fields of research of this PhD thesis were defined within the TeamCom project. The TeamCom project is the umbrella for this research project. Therefore, the described TeamCom approaches are part of this PhD work and are described in detail in chapter 5.

The architecture of the TeamCom framework is presented in Figure 3.25

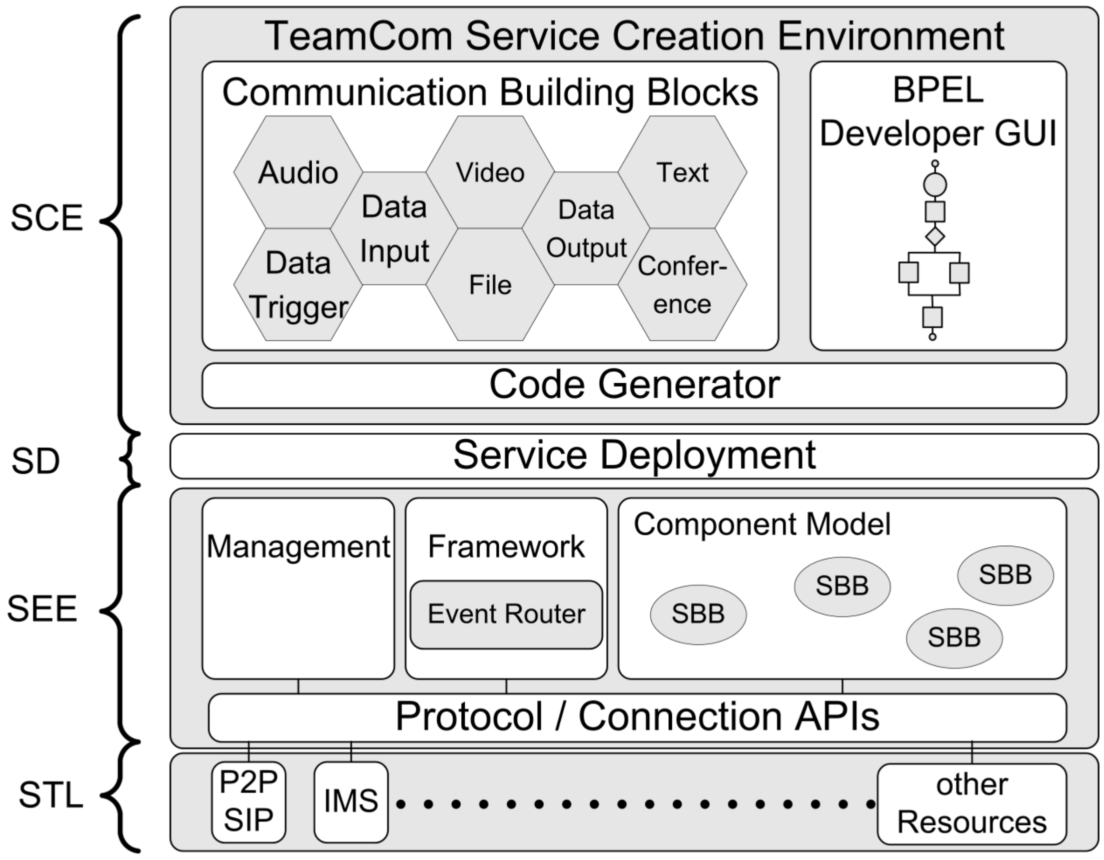


Figure 3.25: TeamCom architecture (Eichelmann, 2010)

The framework consists of the four parts Service Creation Environment (SCE), Service Deployment (SD), Service Execution Environment (SEE), and the Service Transport Layer (STL).

The SCE offers a GUI-based service development. BPEL (refer to section 3.6) is used for the description of the value-added services. The service logic is described using BPEL activities.

Reusable building blocks called "Communication Building Blocks" (CBB) are defined (Table 3.1), which describe the service communication or other functionalities, e.g., video conferencing, database access, or file system operations. The CBB concept of the TeamCom project defines eight pre-defined CBBs. The CBBs are categorised according to the media type utilised. The idea of this kind of categorisation is to define all possible functionalities that are required for value-added services. The CBBs offer an abstraction from the protocol communication within the STL. In BPEL, the CBBs are represented as BPEL partner links.

Table 3.1: TeamCom CBB overview

CBBs	**Functionalities**
Audio	The Audio CBB handles all kinds of audio communication including the establishment of a call, answering calls, manipulation of audio streams (e.g., mixing, transcoding), as well as sending and receiving DTMF tones.
Video	The Video CBB is responsible for playing and recording video streams. It is able to create and close video calls and combine (or mix, transcode) different video signals for merging a new video stream.
Text	This CBB exchanges text messages between two partners and has the capabilities of handling strings, e.g., search for a specific word in a text, replace alphabetic characters or change the encoding of a text.
File	The File CBB handles creation, deletion, sending and receiving of binary files. Another task of File is to write and read any kind of data from and to any position in a file. Finally, this CBB is able to rename files or directories.
Data Input	The Data Input CBB processes all kinds of data queries. This includes database queries as well as reading data from sensors.
Data Output	The counterpart of Data Input is Data Output which is used for writing data to a destination, e.g., to a database, or for controlling an actuator.
Conference	This is a special kind of communication CBB, as it re-uses internal functions of the previously described CBBs, e.g., audio stream mixing. On top of this, the Conference CBB provides functionalities for creating and deleting conference "rooms", as well as adding and

	removing users to/from a room.
Data Trigger	The Data Trigger is closely related to an event generator. If a specific data trespasses a value, an event is triggered. This data can be a sensor value, a timestamp, or periodical dates.

The services are executed in the SEE, which is based on JAIN SLEE (refer to section 4.4). Therefore, the developed BPEL service descriptions have to be translated into JAIN SLEE services (Lasch, 2009a), (Lasch, 2009b). For this purpose, the Code Generator (refer to section 5.3.1) is required. The Code Generator builds the Java classes and the required descriptor files and puts everything into a deployable unit (DU). The DU can be deployed on the JAIN SLEE AS. This step is done by the service deployment.

The service structure, which is used by TeamCom, is the “Single SBB concept” service structure as it is described in section 5.4.1. In this approach, only one monolithic JAIN SLEE SBB is utilised for the whole JAIN SLEE service. The “Single SBB concept” does not support the parallel execution of the service logic, because JAIN SLEE does not support multithreading within the SBB; furthermore the generated SBBs are not reusable for further services.

The TeamCom prototype has proven to be able to describe value-added services with BPEL and to generate executable value-added services from the BPEL service description. However, it also become obvious that value-added services requiring parallel processing cannot be created with the “Single SBB concept”. Therefore, different service structures were analysed. The “Parallel Program Flow concept” (refer to section 5.4.2) supports parallel execution. In this approach, several SBBs are created from the BPEL service description. For each parallel service part, one new SBB is generated that realises this parallel service part. This approach was implemented in the TeamCom prototype. With this new implementation, the TeamCom prototype supports parallel program execution.

The service structures (Eichelmann *et al.,* 2011) which are used in the TeamCom project have some disadvantages, as they are inflexible, monolithic, and tightly coupled service components. A code generator is required for generating the DUs. The services are not easily expandable. Third-party development of service components is difficult, and a reconfiguration at runtime is impossible. To overcome the described problems, novel approaches were analysed in chapter 5.

Responsibilities within the TeamCom project

The author of this PhD thesis was also responsible for parts of the research in the TeamCom research project. Some concepts that are evaluated in this thesis were developed within the TeamCom project. The TeamCom project was the umbrella for the PhD work, and parts of the research work presented in this thesis were carried out by the author within the TeamCom project.

BPEL was chosen as service description language for the TeamCom project. The definition how BPEL is used as service description language was part of the author's responsibilities within this project.

A categorisation of eight different classes of services (audio, video, text, data trigger, data input, file, data output, conference) was defined by the research team of the TeamCom project. These classes of different service categories were called "communication building blocks" (CBBs). The CBBs used by the TeamCom project and the CBBs used in this research work address different problems. The CBB concept described in this thesis was researched by the author. This concept describes a middle layer for the mapping between the implementation of the functionality and the description of the functionality and offers further features which are described in section 6.2.1.

The service creation concept, called “code generator” (cf. section 5.3.1), was researched by the TeamCom research team during the TeamCom project. Since it had certain disadvantages, it was not chosen for this PhD work. Instead of this concept, the runtime service composition concept (refer to section 5.3.2) was selected, which had not been used by the TeamCom project.

The service execution approach, i.e. the “single SBB concept” (refer to section 5.4.1), was researched by the TeamCom research team within the TeamCom project. Because of its disadvantages, it was not applied to the TeamCom project. Instead of this concept, the “parallel program flow concept” (refer to section 5.4.2) was chosen for the TeamCom project. This concept had been previously researched and analysed by the author if this study within the context of his PhD research. The other two concepts the orchestration concept and the choreography concept, were not used by the TeamCom project.

Orchestration in Web Services and Real-Time Communications

In (Lin and Lin, 2007), the authors describe an approach which enables a service creation environment for complex (orchestrated) real-time communication services through a service broker on top of Next Generation Networks (NGN). The goal of their research is to combine the emerging web/telecommunications service spaces with each other. For this purpose, the authors explored three languages that are able to orchestrate workflows: BPEL (refer to section 3.6), CCXML (refer to section 3.4) (W3C, 2011), and SCXML (W3C, 2013). Regarding converged voice-data applications, they take a hybrid approach by encapsulating real-time communication flows, described in CCXML or SCXML, into web services. These web services can be orchestrated in BPEL along with other web services. The authors developed an approach by which communication services can be composed with BPEL.

They conclude that BPEL was more suitable for orchestrating coarse-grained services (refer to section 2.5), whereas CCXML and SCXML were better for fine-grained services. However, using BPEL in combination with web services will work for coarse-grained services only, but will lead to performance problems in case of fine-grained web services. The approach that is followed up in this thesis requires fine-grained service components for the implementation of the value-added services. The approach described in the paper will lead to performance problems; therefore, it is not a solution for this thesis.

StarSCE

In (Baravaglio *et al.,* 2005), the authors analyse the benefits and drawbacks of the web service paradigm applied to a Telco environment using web service composition. As result of the analysis, they found that the orchestration of web services fits to Business Process-oriented services that have no real-time requirements and do not rely on asynchronous interactions. However, many telecommunication web services have strong performance requirements, such as low latency and high throughput. Web services lack in supporting the asynchronous interactions that are required by Telco components. In addition, BPEL4WS, as described in section 3.6, does not natively support all patterns that are relevant for telecommunication web services (Venezia *et al.,* 2006) (Falcarin, 2003a).

Therefore, the authors built an event-based SLEE called "StarSLEE" and a graphical SCE called "StarSCE" in order to develop value-added services. For the description of the services, they use an XML-based language and offer a pre-defined list of service functionalities, which are exposed as web services: third-party call, multi-media conference, messaging, presence, and user's provisioning. The provided services can use different protocols depending on the capabilities of the user devices. In (Venezia *et al.,* 2006) a more detailed description of the StarSLEE and StarSCE frameworks is given. An overview of the StarSLEE communication server is given in Figure 3.26.

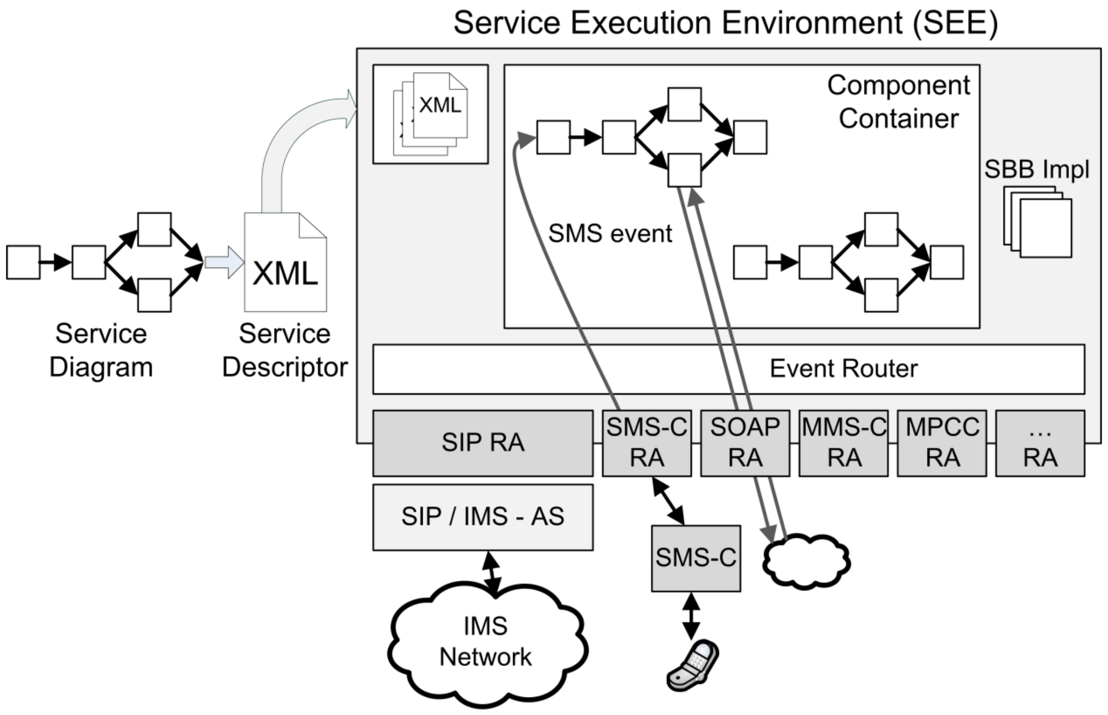


Figure 3.26: StarSLEE communication server (Venezia *et al.*, 2006) © 2006 IEEE

The StarSLEE platform consists of a service execution environment that is an implementation of the JAIN SLEE specification. A SIP RA, a SOAP RA, and other RAs depending on the services are available for the platform. The service descriptor, defined in the StarSCE service creation environment, allows the developer to choose the required SBBs and link them together. This can be done using a graphical development tool. An example of a graphical representation of a service is shown in Figure 3.27. (Venezia *et al.,* 2006)

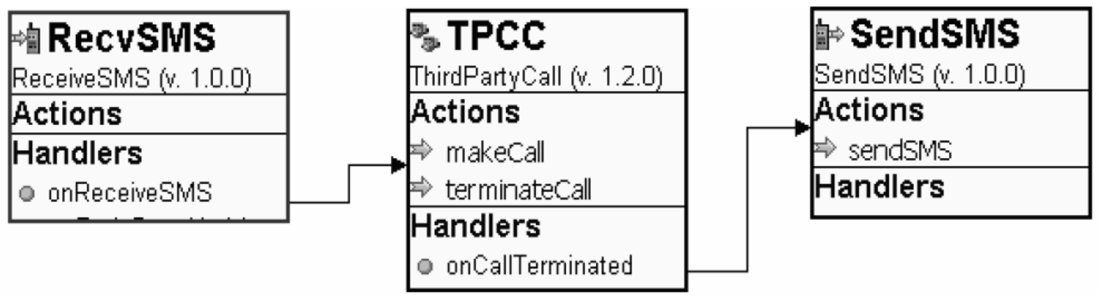


Figure 3.27: StarSCE service description (Venezia *et al.,* 2006) © 2006 IEEE

All elements of the service are implemented as SBBs. Two kinds of SBBs exist, "core SBBs" and "connector SBBs". The "core SBBs" represent the logic of the service. The functionalities third-party call, multi-media conference, messaging, presence, and users provisioning, are available as "core SBBs".

The "connector SBBs" represent elements for the communication with external entities. The "connector SBBs" can be distinguished into "service heads" and "service tails". "Service heads" are SBBs that receive events from external entities (RAs) and send events to "core SBBs" or to "service tails". "Service tails" are SBBs that receive, on the one hand, events from "core SBBs" or from "service heads" and, on the other, send events to the external entities.

For example, in Figure 3.27 the RecvSMS is a “service head” element, the TPCC is a “core SBB” element, and SendSMS is a “service tail” element. Receiving an event in the RecvSMS element triggers the service. An instance of the service is created and the service is executed. In this case, a third-party call control is triggered by a SMS. To make the core functionality of a service available in another protocol, the “service head” and/or the “service tail” can be changed with a “service head” and/or “service tail” that supports the required protocol.

To transform the developed value-added services into communication web services, a web server with a SOAP server is added to the StarSLEE architecture (Figure 3.28). A JAIN SLEE SOAP RA communicates between a SLEE service and the corresponding web service on the web server. Therefore, each service requires a web service implementation on the web server.

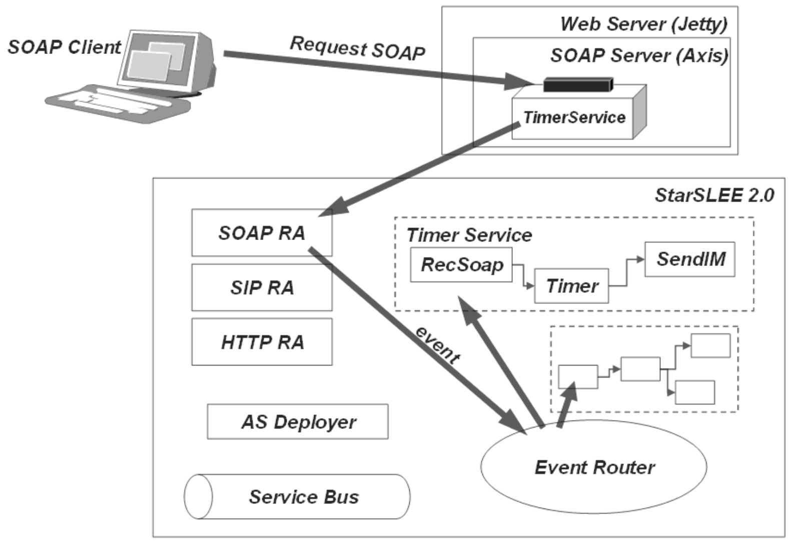


Figure 3.28: STAR-SLEE architecture (Venezia *et al.*, 2006) © 2006 IEEE

StarSCE offers the possibility to develop services based on the functionalities defined within the “core SBBs”, and supports the protocols defined by the “connector SBBs”.

The disadvantage of this approach is that services can only use coarse-grained functionalities in the “core SBBs” and combine them with the “connector SBBs”. The SCE allows orchestrating pre-built services but does not allow defining fine-grained services. Therefore, the possibilities of service creation are limited to coarse-grained services. This approach exposes the services as web services. With this possibility, the services can be orchestrated like normal web services (e.g., with BPEL) but also suffer from the described performance problems. Nevertheless, the SCE was not developed to define value-added services. It was developed to orchestrate services. Therefore, this SCE was not chosen for the service creation in this thesis.

3.10 Conclusion

In this chapter, the most relevant technologies related to service creation for value-added services were investigated, and the advantages and disadvantages for each technology discussed.

In the table below (refer to Table 3.2), the evaluation results are summarised. The criteria are: service description with a GUI (Graphical User Interface), the abstraction from the underlying protocols, and the possibility to define a broad range of value-added services.

With the help of GUI editors or web interfaces, users can create services in an easy and manageable way. The most of the described technologies offer possibilities for graphical service development and are suitable for this purpose. The exceptions here are “A SIP-based Programming Framework for Advanced Telephony Applications” and SCML which offer no graphical editor. The graphical user interface of the project "A High Level Service Creation Environment for Parlay in a SIP Environment" only offers rudimentary possibilities for service creation and only allows call-related services.

Table 3.2: Service creation solutions

	graphical service description	abstraction from the underlying protocols	possibility to define a broad range of value-added services
Call Processing Language (CPL)	yes	yes	limited
Language for End System Services (LESS)	yes	yes	limited
Voice XML (Voice Extensible Mark-up Language)	yes	yes	limited
CCXML (Call-Control eXtensible Mark-up Language)	yes	yes	limited
SCML (Service Creation Mark-up Language)	XML/Text editor	yes	limited
WS-BPEL	yes	yes	unlimited
BPMN	yes	yes	unlimited
TelcoML	yes, with UML tool	yes	limited to the TelcoML enabler library
A High Level Service Creation Environment for Parlay in a SIP Environment	yes, but very rudimentary	yes	limited
SPICE	yes	yes	limited
A SIP-based Programming Framework for Advanced Telephony Applications	no	yes	limited
Session Processing Language (SPL)	yes	yes	limited
OPUCE	yes	yes	limited
TeamCom	yes	yes	unlimited
Orchestration in Web Services and Real-Time Communications	yes	yes	limited
StarSCE	yes	yes	limited

The described technologies also abstract from the underlying protocols, so also non-experts can develop services. Since LESS is targeted for end system services, it is not usable for a value-added service that runs on an Application Server.

In combination with the service execution solution (refer to chapter 4), the service description language (refer to chapter 3) should support new protocols and functionalities in order that the developer is able to describe a broad range of value-added services. Here, with CPL, LESS, Voice XML, CCXML, and SCML it is not possible to add functionalities and protocols from other domains. CPL, CCXML, and SCML support only call control functionality. Voice XML only supports voice applications. TelcoML supports both call control and voice applications but is limited to the TelcoML enabler library and allows only coarse-grained service orchestration. BPEL and BPMN are implemented as web services, they support the SOAP protocol. The support of other protocols in the service description is not directly possible. With BPEL and BPMN, it is possible to orchestrate web services which can support other protocols. From the related research projects only SPICE, OPUCE, TeamCom, and StarSCE, support new protocols in the service description. Only the TeamCom project supports a wide range of possible value-added services. The problem with TeamCom is that the adding of new protocols is very complex.

Principally, both BPMN and BPEL can be used to describe a wide variety of value-added services, but BPEL was specially designed for the description of executable processes. Hence, BPEL will serve this purpose better. Furthermore, BPMN can be translated into BPEL.

However, BPEL will only be used as service description language and not in combination with a BPEL engine. Using a BPEL engine to orchestrate the service components will work for pre-defined basic coarse-grained services. This was the case in most of the related research projects (refer to section 3.9 and section 4.8). Using a BPEL engine for orchestrating fine-grained services would require implementing the services as web services, which again would lead to performance problems.

Altogether, BPEL fulfils the required criteria and offers the most possibilities for the description of value-added services. Because web services should not be used, another concept was researched to support new protocols. A possibility of how to add the support of new protocols to BPEL is described in section 5.2.3.

4 Current Solutions for Service Execution and Provisioning

The created value-added services have to be executed on a service execution environment (SEE). The focus of this chapter is the actual provisioning and execution of value-added services. The technologies are evaluated using the criteria introduced in chapter 2.

The relevant investigation criteria described above are: supported protocols, performance, different service possibilities, and composition capability/reusability.

The next sections will provide a description of the relevant technologies; thereafter, related research projects will be introduced.

The specific technology JAIN SLEE will be described in more detail, since it will be used as the basis for the prototype implementation presented in chapter 6.

4.1 Customised Application for Mobile Network Enhanced Logic Service Environment (CAMEL SE)

CAMEL SE (CSE) (3GPP, 2014) is based on CAMEL IN (Customized Application for Mobile network Enhanced Logic – Intelligent Network) services in mobile networks. To execute services that are provided by the CSE in NGN/IMS (Next Generation Network/IP Multimedia Subsystem), a protocol conversion is required. This conversion is provided by the IM-SSF (IP Multimedia-Service Switching Function). An overview of the CSE architecture with the IM-SSF is given in Figure 4.1. Here, the two protocols SIP (on the ISC interface, IMS Service Control) and CAP (CAMEL Application Part) are translated into each other.

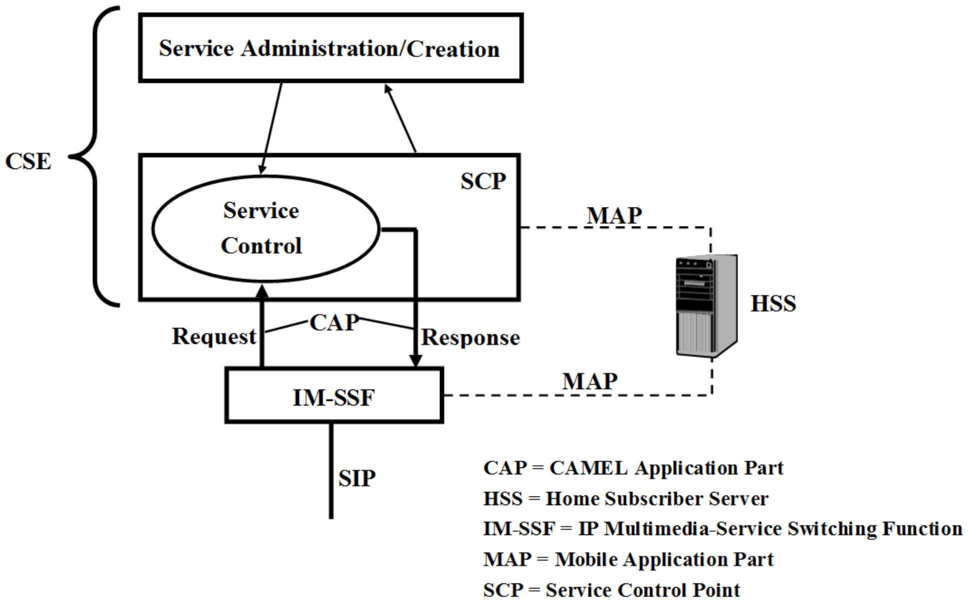

Service Administration/Creation
CSE
SCP
Service Control
MAP
HSS
Request
CAP
Response
IM-SSF
MAP
SIP
CAP = CAMEL Application Part
HSS = Home Subscriber Server
IM-SSF = IP Multimedia-Service Switching Function
MAP = Mobile Application Part
SCP = Service Control Point

Figure 4.1: CAMEL Service Environment (CAMEL SE) (Detecon, 2007)

The CSE consists of two components, the SCP (Service Control Point) and the SCE (Service Creation Environment). The SCP manages and activates the available services. The SCE is based on modules, which are called SIBs (Service Independent Building Blocks). With the help of these SIBs, services can be created.

Typical services of the CAMEL-IN are, e.g., call rerouting and televoting. The advantage of this technology is its ability to re-use already developed CAMEL-based services. The development costs of new services using this technique are too high and therefore not recommended; there are other techniques available which are more cost-efficient. (Detecon, 2007)

Moreover, IN-based technologies do not provide the desired level of flexibility in service provisioning. The service platform is limited because of its direct interworking with the underlying network protocols and switching equipment (Magedanz, 2006). This makes IN service development difficult which requires specialised telecommunication knowledge. Another limitation is that CAMEL is bound to the legacy telecommunication services and is not suited for the emerging multimedia services (Magedanz, 2006). On the one hand, CSE shows good performance due to its direct interworking with the network protocols, but the number of supported protocols is rather limited.

4.2 OSA/Parlay, Parlay X

The industry consortium Parlay Group was founded in 1998 with the aim of specifying APIs (Application Programming Interfaces) that simplify and unify the access and control of telephone networks. Within 3GPP, the Parlay specification is part of the Open Service Access (OSA) architecture and is therefore called OSA/Parlay (Parlay, 2010).

OSA/Parlay provides access to network functions that are offered as service capability features (SCF) through service capability servers (SCS).

Basically, OSA/Parlay defines client applications (CA), service capability servers (SCS) providing service capability features (hiding the telecommunication networks), and a framework. The SCSs provide standardised interfaces offering access to the network functions, i.e. service capability features (Figure 4.2).

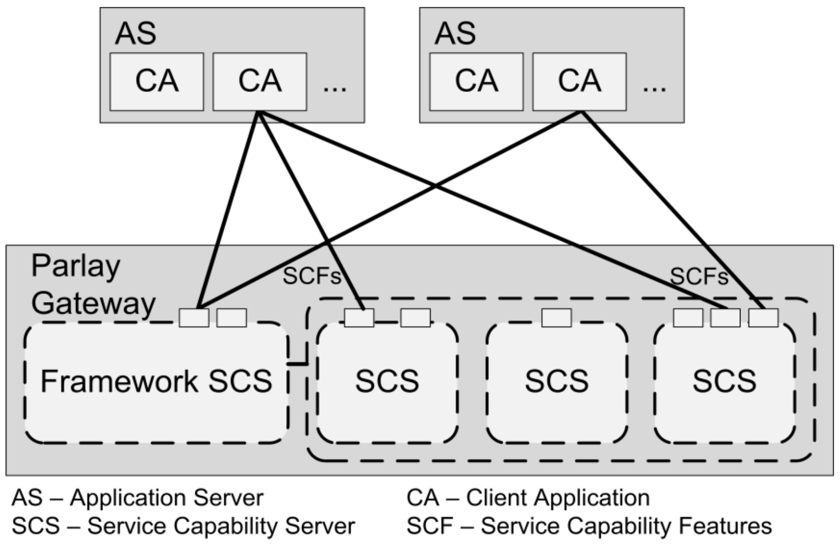


Figure 4.2: OSA/Parlay framework (related to (Abarca *et al.,* 2002))

The OSA/Parlay Gateway consists of several SCSs. One of the SCSs is called the "Framework" (Abarca *et al.,* 2002).

There are two possibilities for implementing the CAs (Figure 4.3):

- using low-level programmatic Parlay APIs of a Parlay Gateway or
- using higher-level web services offered, e.g., by a Parlay X Gateway.

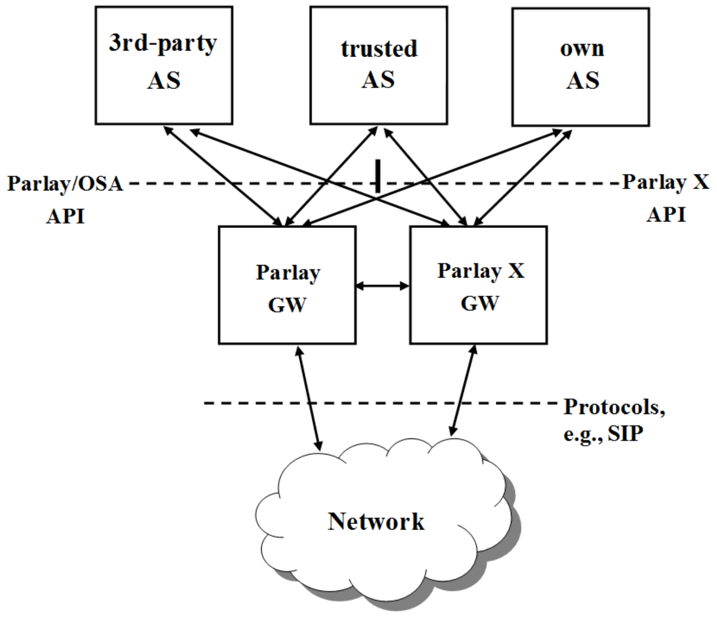


Figure 4.3: Parlay gateway (related to (Detecon, 2007))

The gateways can protect the access of the service capability features against the client applications so that 3rd-party applications are allowed, too.

The Parlay techniques are specified in UML (Unified Modelling Language) (OMG 2011a) format. Several standard interfaces and gateways are defined, but no application servers (ASs). A Parlay/OSA Gateway translates protocols such as SIP into the so-called "OSA API". The application server can be addressed with CORBA-based (Common Object Request Broker Architecture) (OMG, 2012a) interfaces. The gateway (Figure 4.3) itself serves as middleware to provide a secure and abstract access to network functions for the implementation of telecommunication services.

The Parlay specification consists of three parts: (i) the framework and service capability features, (ii) the OSA/Parlay functionality, and (iii) the framework functionality.

The framework and the service capability features are connection points to the network functionality of the telephony network. They offer, e.g., the possibility of initiating, controlling, and stopping a call and playing an announcement.

The OSA/Parlay functionality describes services that run on special application servers. These services implement the service logic and use the service capability features of the telephony network.

The framework functionality offers, for instance, the functionality for authentication and identification of the service against the service capability features and is responsible for giving access permissions for these capabilities to the services. It also allows the interoperability of service capability features among different providers and services.

An advantage of this technique is the linking of a third-party application server to an NGN/IMS in a secure way, because OSA itself offers discovery, authentication, registration, and access control. The alternative solution, Parlay X, is based on web services technology. With the Parlay X solution, the web services can be used to realise an open access to NGN capabilities. Both techniques can be used in combination. The advantages of these two solutions are the high-level of security, the possibility of combining web services using Parlay/OSA functions, the possibility of adding the support of new protocols through resource adaptors, its expandability, and the support of several programming languages (C++, C#, Java). The framework was specially developed to support telecom applications and offers a good performance for these services; third-party development is possible, too.

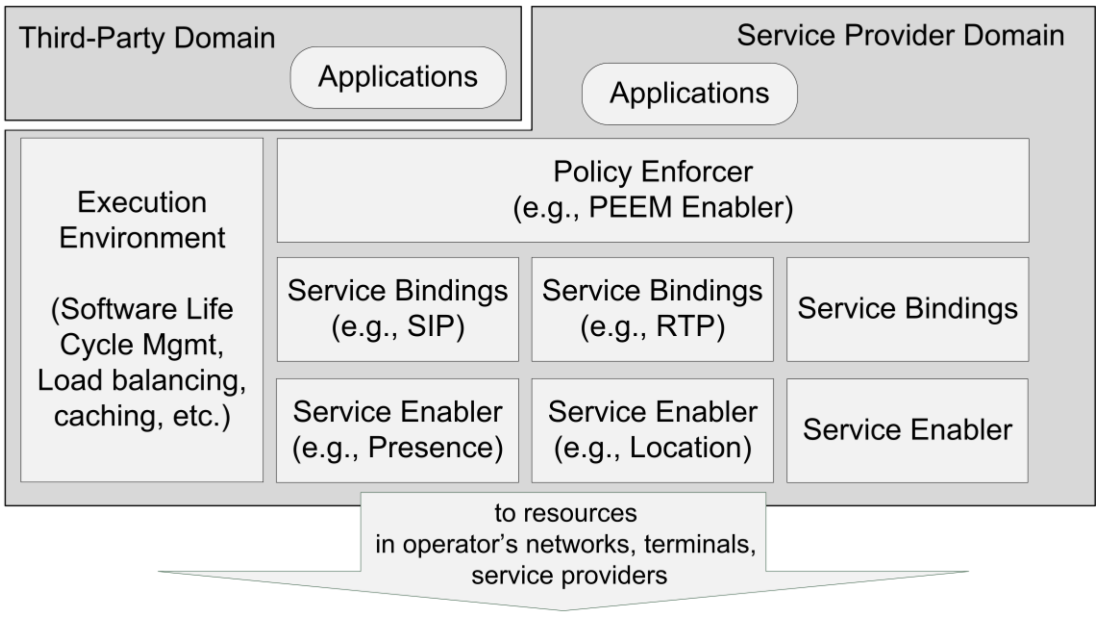
OSA/Parlay and Parlay X specify open interfaces offering service capability features, but do not specify an execution environment for value-added services. Therefore, this approach can only be part of the solution.

4.3 OMA SE (Open Mobile Alliance Service Environment)

The Open Mobile Alliance (OMA) (OMA, 2013) specifies open global standards for network-independent applications and service components, especially for cellular mobile networks. The main requirements are the independence of operating systems, execution environments, programming languages and vendor platforms, as well as interoperability between devices and across networks (roaming), between infrastructure and service providers. Therefore, various technologies like IMS, Parlay, or web services can be used for an implementation of OMA specifications.

In Figure 4.4, an overview of the OSE architecture is shown. It consists of the Service Enabler, the Policy Enforcer, applications, the execution environment, the Interface Bindings, and the PEEM (Policy Evaluation, Enforcement, and Management).

OMA mainly defines so-called "Service Enablers" and applications within the OSE reference architecture. An example of a Service Enabler is the presence function. OSE specifies how the Service Enablers work together and how they provide their resources via standardised interfaces with the help of the Interface Bindings. OSE offers an easy, safe, and secure access to network resources. The applications realise the communication services by utilising the Service Enablers. Hence, a service can be realised, e.g., by using application servers within the OSE or outside the OSE ("third-party"). The Policy Enforcer offers security rules among applications and Service Enablers, and among several Service Enablers. The PEEM can be used as a central Service Enabler who allows other Service Enablers to show their functionalities. The execution environment handles aspects like service life cycle management, load balancing, or caching (Detecon, 2007).


Third-Party Domain
Applications
Service Provider Domain
Applications
Execution Environment
(Software Life Cycle Mgmt, Load balancing, caching, etc.)
Policy Enforcer
(e.g., PEEM Enabler)
Service Bindings (e.g., SIP)
Service Bindings (e.g., RTP)
Service Bindings
Service Enabler (e.g., Presence)
Service Enabler (e.g., Location)
Service Enabler
to resources in operator's networks, terminals, service providers

PEEM – Policy Evaluation, Enforcement, and Management
WS – Web Services
RTP – Real-Time Transport Protocol
SIP – Session Initiation Protocol

Figure 4.4: OMA SE architecture (related to (OMA, 2004))

The advantages are a high degree of safety, the usage of different technologies like web services or Parlay, and the independence of the programming language. New protocols can be supported by adding new interfaces bindings. OSE was specially developed for telecommunication services. The performance depends on the implementation. The disadvantage is that the services primarily cover the mobile area (Detecon, 2007).

4.4 JAIN SLEE (Service Logic Execution Environment)

In March 2004, the JAIN SLEE (Java API for Intelligent Networks/Service Logic Execution Environment) specification was introduced as JSR 22 (Java Specification Request) (Sun and Open Cloud, 2004) into the Java Community Process (JCP) with Sun Microsystems, Open Cloud, Fujitsu Siemens, IBM, 8x8, Motorola, Nortel Networks, NTT, Personeta, Telcordia Technologies, TrueTel, Siemens and Vodafone involved in the standardisation. The JAIN SLEE specification (Sun and Open Cloud, 2004) defines a Java-based and component-based runtime environment that is designed specifically for scalable, asynchronous event processing based on concepts similar to the Java EE (Java Platform, Enterprise Edition), but explicitly designed for supporting intelligent networks in the telecommunication industry. In this thesis, JAIN SLEE is used for a prototype implementation and is therefore presented in more detail.

The main goal of the JAIN SLEE development was to achieve low latency and high throughput, both required by communication networks, aiming to provide a response time below 200 ms and processing of thousands of events and transactions simultaneously as well as achieving 99.999% availability. The specification defines a container model and components called Service Building Blocks (SBBs). Based on these components, the JAIN SLEE specification uses proven concepts from Java EE and allows the decoupling of services from underlying networks through resource adaptors. Overall, JAIN SLEE does not replace the Java platform, but is a complementary platform for the requirements of the telecommunication industry. In the JAIN SLEE standard, an integration of Java EE and JAIN SLEE applications is described (Sun and Open Cloud, 2004).

Since July 2008, the version 1.1 of the JAIN SLEE specification has been published as JSR 240 (Sun and Open Cloud, 2008). This specification is mainly an extension of the first version with focus on the development of resource adaptors and their architecture.

According to (Sun and Open Cloud, 2004), JAIN SLEE is an application server. An application server (AS) provides an environment where applications can run. It provides services to the applications and offers management and/or developer tools. Furthermore, it can distribute requests across multiple physical servers and provides a container model for applications (Ottinger, 2008).

The basis of the JAIN SLEE architecture contains four areas: management, framework, component model, and resource adaptors/APIs. Figure 4.5 gives an overview of the JAIN SLEE architecture.

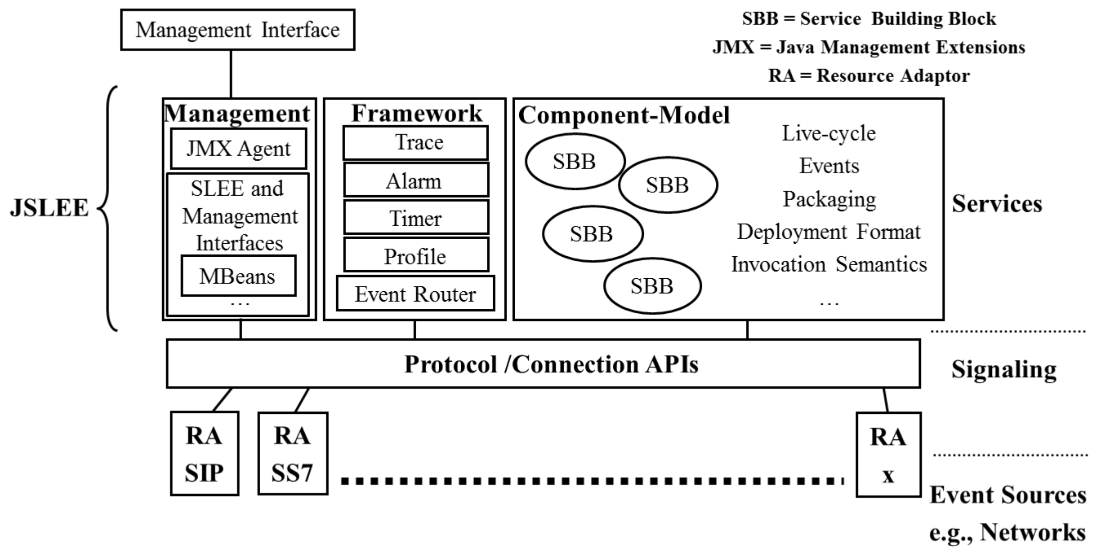


Figure 4.5: JAIN SLEE architecture (related to (Maretzke *et al.,* 2005))

Management

For the management of the JAIN SLEE environment, the Java Management Extensions (JMX) technology can be used, which was developed in the JCP (JSR 3, JSR 160, JSR 255) with the participation of companies such as IBM, BEA Systems, and Borland. With JMX, a framework is provided which allows the developers to implement management capabilities in Java and integrate them into their applications. Many server applications are currently using the JMX specification. The control of a resource, which is manageable with JMX, is realised by Managed Beans (MBeans).

The JAIN SLEE environment is also managed through standardised MBeans that control the runtime environment, the installation of JAIN SLEE elements, and the management of services. Furthermore, the MBeans offer service usage statistics, and the data supply of services via profiles. With the help of the MBeans, a graphical user interface can be implemented for administration tasks.

Framework

The elements of the framework in the JAIN SLEE architecture form the basis of the actual service logic: timers, alarms, traces, profiles, and the event router. A timer triggers the service logic at specific time events. By alarms, external management systems are notified. Traces allow the output of messages (e.g., logging). With profiles, data related to user and service profiles are stored and managed. The most important part of the framework is the event router which forwards received and generated events to the appropriate SBBs or resources.

Resource Adaptor

Resource adaptors are elements of the JAIN SLEE architecture that enable the communication with networks, systems, or databases outside the SLEE. The required protocol APIs for communication are implemented in the resource adaptors. In the SLEE environment, resource adaptors can be installed and used simultaneously. If the SLEE environment receives a specific signal (e.g., a protocol message) from an external source, the resource adaptor is converting this signal into simple Java objects. These Java objects, which, in JAIN SLEE, are also called events, extract all relevant information from the source and transmit this information to the SBBs. The SBBs can register for the events they want to receive. If more than one SBB is registered for an event, then the event is forwarded to all of these SBBs.

Component Model

The component model is a core part of the JAIN SLEE architecture and controls the usage of components, i.e. service building blocks (SBBs) in the SLEE environment. It fulfils the following tasks:

- interaction of SBBs with each other and with the SLEE environment;
- execution of the components;
- packaging of services and components in JAR archives and deployment;
- configuration using deployment descriptors.

The SBBs in JAIN SLEE follow a life cycle that is controlled by the surrounding run-time environment, similar to the Enterprise JavaBeans (EJBs) in Java EE. This environment is responsible for managing the event processing of the SBBs and the calls of the framework. Transactions are used for the event processing and the calls of the framework. This guarantees that the JAIN SLEE container is always in a consistent state, even if an error occurs.

Event Model

An event can be initiated by a signal source from outside or from inside the SLEE environment. Figure 4.6 illustrates the JAIN SLEE event model.

Events that occur within the SLEE environment are usually produced by SBBs or framework elements, e.g., the timer facility, in order to send signals to other components or to communicate with them. When an event producer sends an event, the event type has to be known by the SLEE environment. This event type defines how the event will be routed by the SLEE environment and which SBBs will receive the event (Sun and Open Cloud, 2004). The event model of JAIN SLEE is based on the publish/subscribe model. This means that the senders (publishers) themselves do not send their events to specific recipients.

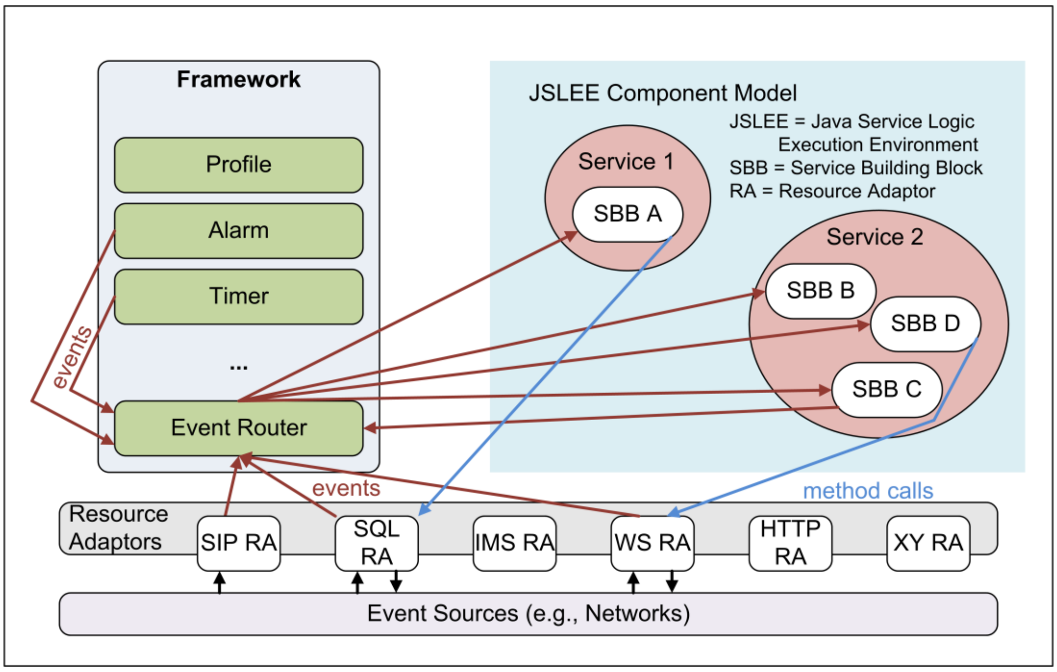


Figure 4.6: JAIN SLEE event model (related to (Maretzke, 2005))

Instead, the recipients (subscribers) have to register/subscribe for the events. The events are sent to a central point from where they are delivered to all recipients (subscribers) who have been registered/subscribed for the events (Sun and Open Cloud, 2003). This central point is called "event router".

The JAIN SLEE specification defines the concept of the activity context (AC), which maintains the relationship between event producers and event consumers, to implement this model. For example, the AC concept allows the processing of subsequent events that are assigned to the activity context. A simplified example of the event processing in the SLEE environment is illustrated in Figure 4.7.

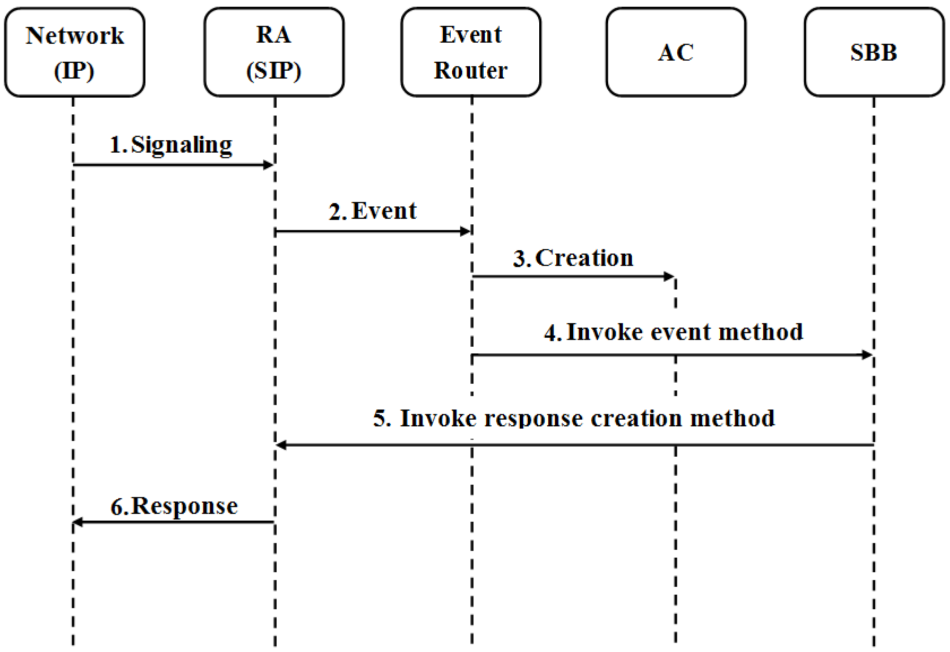


Figure 4.7: Example of the event processing in JAIN SLEE ((related to Maretzke *et al.,* 2005))

The process starts with a network-generated signal that is forwarded to the resource adaptor (1); this may be a signal for setting up a call. The resource adaptor converts the incoming signal to a Java event and transmits it to the event router (2). Since this event is the first one in a sequence of events to establish a telephone call, the event router creates a new activity context (3), then the SBB component passes the event and the activity context as parameters to the event processing (4). The service logic is invoked, and the SBB calls specific methods (5) that are offered by the resource adaptor to generate a response to the network (6). Upon the execution of the service logic, the SBB will detach from the AC, then the AC will be destroyed. This example, provided by (Maretzke *et al.,* 2005) describes the activity context in combination with the event processing in JAIN SLEE. The event router is always involved in the event processing. In the following chapters, the visualisation of event routing and activity contexts is simplified and the event router is not illustrated anymore but in reality, it is always involved.

Service Building Blocks (SBBs)

Service Building Blocks (SBBs) are software components that can send and receive events. SBBs also include the service logic that is executed, depending on the type and status of the incoming event. JAIN SLEE services and SBB components can be differentiated from each other as follows (Sun and Open Cloud, 2003):

- An instance of a JAIN SLEE service can consist of a single SBB or can contain multiple instances of different types of SBBs.
- The same SBB can be included in several service types.
- An SBB can only execute one event at the same time.
- Several SBBs that belong to the same JAIN SLEE service can process events in parallel.

An SBB consists of the SBB descriptor that describes the component in XML and the implementation of an abstract SBB base class. The developer must extend this abstract class for each SBB and implement an event handler method for each event that can be received by the SBB. The content of these methods represents the current logic of the service (Haiges, 2005). The SBB descriptor of an SBB component includes all information required for the event router to deliver the events to the interested SBBs:

- name of the SBB;
- name of the vendor of the SBB;
- SBB version;

- list of events that can be received or sent by the SBB component;
- names of the Java classes that implement the service logic of the SBB component.

Similar to the deployment of EJBs in Java EE containers, several descriptors and special structures must be created for SBBs so that they can be installed as new components in the JAIN SLEE container. The elements that are required for the service can be packaged in a deployable unit (DU). The DU is packed in a Java Archive (JAR) file that can be deployed into the SLEE. The DU can contain services, SBBs, events, profile specifications, resource adaptors, resource adaptor types, library files, and the deployment descriptor. This descriptor mainly includes references to all those service component or resource adaptors that can be installed into the SLEE. These elements are also packaged in form of JAR.

Each SBB is packaged in a separate JAR archive that contains the compiled Java SBB class and the SBB descriptor. The example DU shown in Figure 4.8 consists of two SBBs, the SBB A in the file a_sbb.jar, and SBB B in the file b_sbb.jar. Each SBB requires its own SBB descriptor (here a_sbb-jar.xml and b_sbb-jar.xml) and its SBB class ("a_sbb.class" and "b_sbb.class"). The SBB descriptor references the SBB class and the events in which the SBB is interested. This information is of central importance for the event router that delivers the events to the interested SBBs. The service descriptor (service.xml) contains information about the service, the contained SBBs and the SBB hierarchies. With JAIN SLEE, it is possible to build complex hierarchies of SBBs, e.g., for re-use. In such cases, the order of the delivery of events must be regulated by priorities in the service descriptor. The deployable unit describes the classes, the profile, and the events that are used in the service. (JSR 240, 2008)

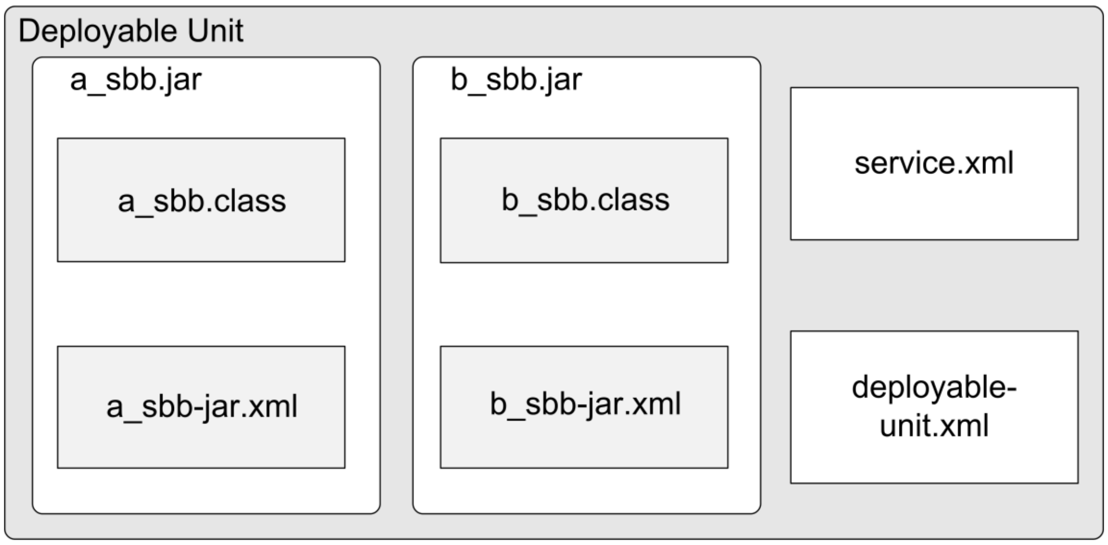


Figure 4.8: Elements of a deployable unit

With the following example service, the event processing within JAIN SLEE is discussed in more detail. The example service is a chat service. Multiple chat clients can connect to this service. When the service receives a chat message from one of the chat clients, it sends out this message to the other chat clients who are connected. An overview of the required service components is presented in Figure 4.9 and an extract of the Message Sequence Chart (MSC) for this example is shown in Figure 4.10. The sessions are established with the SIP protocol, and the user data is exchanged encrypted via TLS (Transport Layer Security) connections. The first step is the establishment of the SIP session. Within the SIP session, the TLS user data connection can be established, and the user data can be exchanged encrypted via TLS.

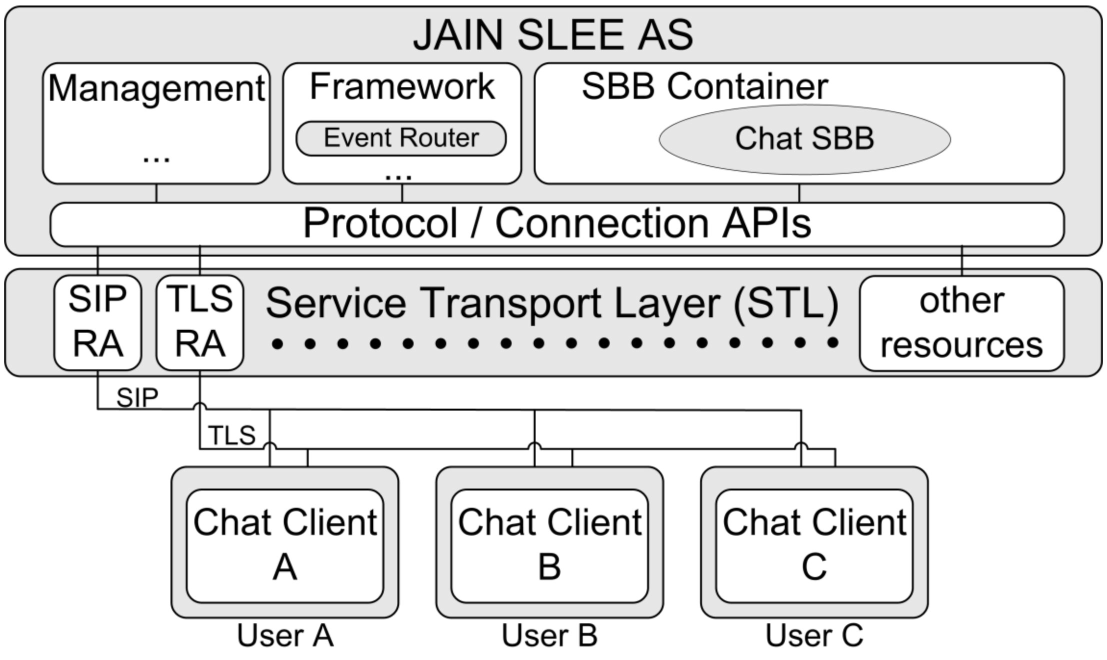


Figure 4.9: JAIN SLEE components for the TLS-chat service

The service requires one SBB (Chat SBB), which contains the service logic, and two resource adaptors (RAs), the SIP RA and the TLS RA. The SIP RA offers the functionality to handle the signalling part for the service and offers specific SIP functionalities to the SBBs. The TLS RA handles the user data part for the chat service. It offers the functionality to handle a TLS connection and transfer encrypted user data via the TLS protocol. The users A, B, and C use their chat clients to join the chat. The chat clients have to understand the SIP protocol for the signalling and the TLS protocol to transfer the user data.

The MSC in Figure 4.10 shows the protocol communication between the chat clients and the service on the left and the event communication within the service on the right side.

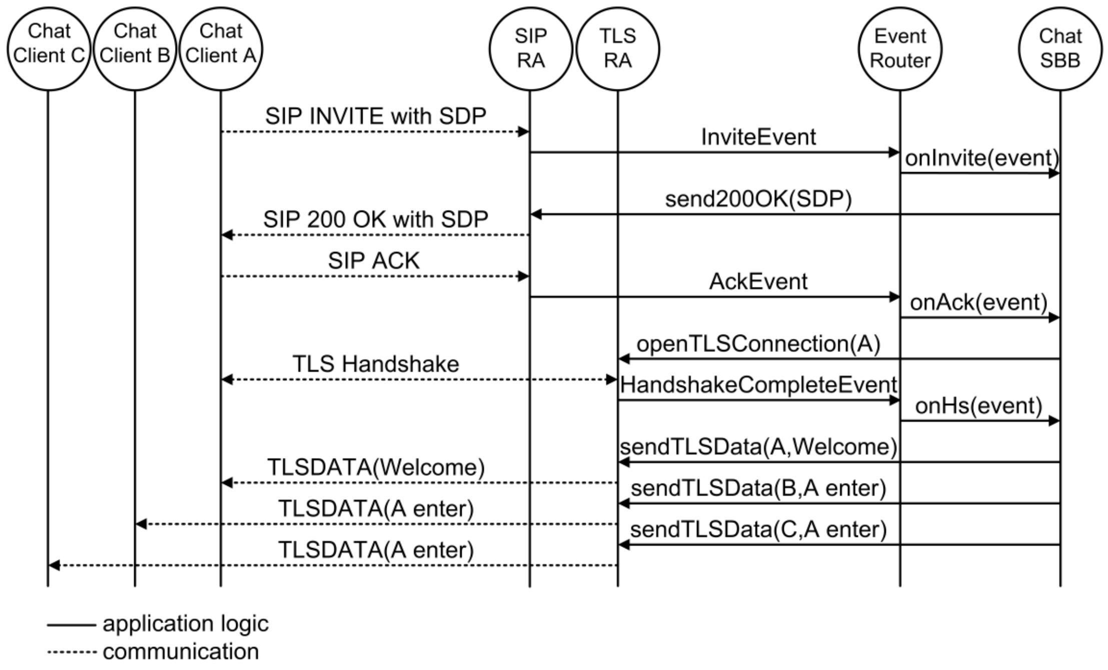


Figure 4.10: Connection establishment with the chat service

The service starts with the establishment of the SIP session between the chat client A and the service with resulting event communication within the JAIN SLEE service. After the SIP session is established, the TLS user data transfer between the service and the chat client is initiated from the Chat SBB with the call of the `openTLSConnection(A)` method of the TLS RA. In the next step, the TLS session establishment and the TLS user data transfer between the service and the chat clients is started.

In the figure, three users wish to participate in a chat. For this purpose, chat clients B and C have already established a SIP session and a TLS connection with the service, they are already logged into the service.

The chat client A initiates the session establishment procedure with a SIP INVITE message. This procedure is called “SIP three-way handshake”. The chat client A sends the SIP INVITE to the SIP RA of the application server.

This INVITE message includes signalling and session information. The session information is described by the Session Description Protocol (SDP) that is encapsulated within the INVITE message. In case of the chat service, the SDP include the kind of application (TLS chat), the IP address, and the TCP port number, at which the chat client is listening.

The SIP RA receives the SIP INVITE, generates an invite event with the required information, and forwards this event to the event router. The event router looks up all SBBs that are interested in this invite event (Chat SBB) and calls the invite event handler methods of those SBBs (`onInvite(event)`). The received invite event is available as input parameter of the event handler method. The `onInvite(event)` method of the Chat SBB analyses the received invite event. In this example, the invite handler method generates a SIP response (200 OK) by calling the according method from the SIP RA (`send200OK(SDP)`).

This response message includes an SDP part within its SIP body, which informs the chat client A about the port for the TLS user data communication. As answer to this response message, chat client A sends a SIP ACK message to the SIP RA of the JAIN SLEE server. With the reception of this message, the three-way handshake is finished and the SIP session is established.

The SIP RA generates an ACK event and sends this event to the event router. There the interested SBBs are identified (here Chat SBB), and the ACK event handler method of each SBB is called (`onAck(event)`). The ACK event is transmitted as parameter of the method to the SBB.

The `onAck(event)` method of the Chat SBB activates the TLS connection establishment by calling the respective method on the TLS RA (`openTLSConnection(A)`). The RA initiates the TLS connection with the TLS handshake.

In case of a successful TLS session establishment, the TLS RA generates the event HandshakeCompleteEvent, which is received by the event router. Again, the event router identifies the SBBs that are interested in this event (here Chat SBB) and calls the event handler methods of those SBBs. In this case, the HandshakeCompleteEvent handler method `onHs(event)` is called. The `onHs(event)` method generates a welcome (login) message for the client A and a notification message for B and for C to inform the clients that A has logged in, by calling the methods for sending TLS data (`sendTLSData(…)`) on the TLS RA. These messages are sent encrypted within the TLS connections to the chat clients.

This example shows the abstraction of the service logic from the protocol. The protocol-specific functionalities are handled by the RA and are offered as methods for the SBBs.

Graphical service development

Some JAIN SLEE implementations offer tools, which support a graphical service development. These tools can speed up the development process and simplifies the service development. A noteworthy graphical development tool is the Rhino Visual Service Architect (OpenCloud, 2013) from OpenCloud. This tool offers a graphical representation of the service. It allows switching between the graphical representation and the textual code. This tool produces parts of the service code. This code can be manipulated and adapted by the developer. The Rhino Visual Service Architect generates code that includes finite state machines for asynchronous design and resource adaptors for specific protocols. Furthermore, it supports templates. These templates provide pre-designed starting points for the creation of a service. (OpenCloud, 2013)

The Rhino Visual Service Architect offers a good support for the service development, but it is required to manipulate and edit the service code, which requires expert knowledge. Therefore, it is not a solution for the SCE of this thesis but a good solution to support the development of the CBBs described in chapter 6.2.1.

Advantages and disadvantages of JAIN SLEE

The advantages of the JAIN SLEE technology are flexibility, platform independence, low latency, and high throughput. The technology has been developed to fulfil the specific requirements for telecommunications. JAIN SLEE is extensible because the technology is based on Java. By developing and adding new resource adaptors, the support of new protocols can be added. Furthermore, platform-dependent programming languages such as C/C++ can be integrated through the Java Native Interface (JNI).

The disadvantages are the required specific knowledge that also makes the development of new resource adaptors and value-added services difficult. The developer must have expertise in Java, JAIN SLEE, and the required protocols. The development of a typical service would take a long time, such as 3 months or even more (Detecon, 2007).

The JAIN SLEE technology fulfils the requirements for the SEE. Unfortunately, the requirements for the SCE cannot be fulfilled (refer to section 2.5). It is also not possible to use a service creation technology directly in combination with JAIN SLEE (refer to chapter 5). However, these shortcomings can be solved with the proposed extension of JAIN SLEE as described in chapter 6.

4.5 SIP Common Gateway Interface (SIP-CGI)

SIP-CGI is a language-independent interface that allows interactions with programs or scripts on an SIP Application Server (IETF RFC 3050, 2001). The advantage of SIP-CGI is the possibility to use all programming or scripting languages, as long as they can be executed on the SIP application server. The data from the incoming SIP messages are passed to the executing programs. SIP-CGI scripts usually have the same access to resources on the server as other server software. For security reasons, only the service provider should create services.

Figure 4.11 shows the SIP-CGI model. The elements in this figure are two SIP servers and two SIP clients. A CGI program is located on one of the SIP servers. When the server receives a SIP request, it can execute the SIP-CGI program with the required parameters. The SIP-CGI program computes an answer for the SIP server. With this answer, the SIP server can modify the received SIP request or generate a SIP response and send it to its destination.

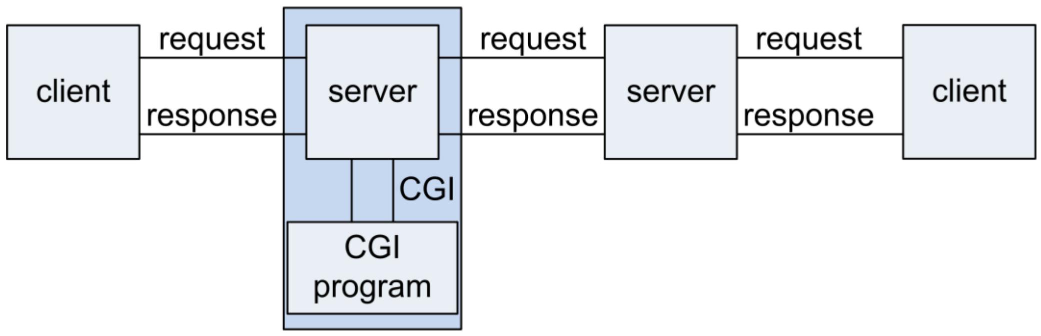


Figure 4.11: CGI model for SIP (related to IETF RFC 3050, 2001)

SIP CGI is an expandable interface with the feature of executing applications and scripts on a server. SIP CGI offers many service possibilities.

Disadvantages of SIP CGI are the relatively low execution speed, especially when scripting languages such as Perl are used, and the security problems, described above. Another disadvantage is that the integration of other protocols and the support of multimedia functionalities are not standardised. All these disadvantages lead to the conclusion that SIP-CGI is not an adequate solution for the SEE in this work.

4.6 Web Services

Legacy web services (W3C, 2004a) are distributed software applications that are based on the service-oriented architecture (SOA) (OASIS Standard, 2006). Web services use standardised interfaces, which are described using the Web Service Description Language (WSDL) (W3C, 2007b) and protocols like SOAP (W3C, 2007c). The Universal Description and Discovery Interface (UDDI) is used as service registry. The legacy web services normally follow the “find-bind-execute” paradigm of SOA (Figure 4.12). This paradigm describes the communication between the service provider, the service registry, and the service requestor “user”.

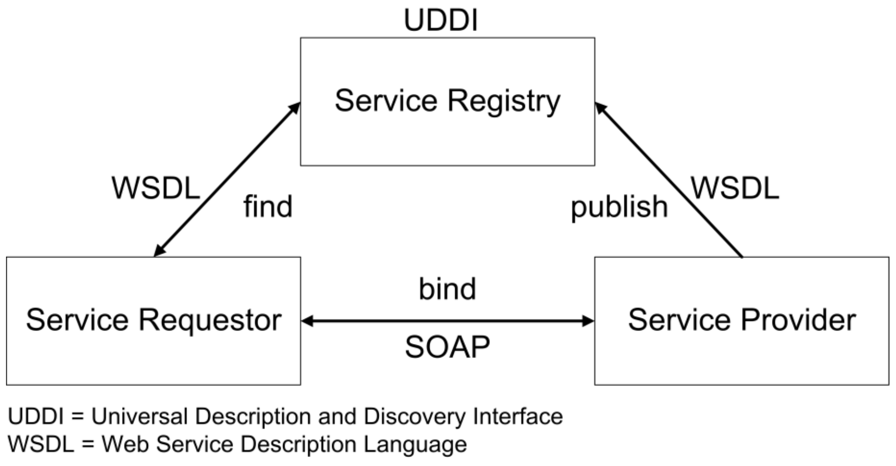

UDDI
Service Registry
WSDL
find
publish
WSDL
Service Requestor
bind
SOAP
Service Provider
UDDI = Universal Description and Discovery Interface
WSDL = Web Service Description Language

Figure 4.12: Web services and SOA

Web services are independent from the programming language, the execution platform, and the transport protocol (e.g., SIP or HTTP). Based on XML messages, web services combine distributed and object-oriented programming standards and they are expandable with nearly unlimited service options. They can be composed with other web services to enable services that are more complex.

The W3C identified two major classes of web services: REST-compliant web services and arbitrary web services. The primary purpose of the services in REST-compliant web services is to manipulate XML representations of web resources using a uniform set of 'stateless' operations. The services in arbitrary web services may expose an arbitrary set of operations. (W3C, 2004a)

Web services tend to provide a poor performance because of the overhead introduced by protocols such as SOAP (Hammerschall, 2005). Web services are specially developed for business process oriented services but they do not fulfil the requirements for real-time (Baravaglio *et al.,* 2005) services.

For IT services this works very well, but for services from the telecommunication domain it becomes slow due to the performance limitations of Java EE-like application servers. These servers are designed for enterprise services based on synchronous request-response interactions, but they do not perform well in a telecommunication service environment using asynchronous interactions. Web Services have been developed for composing, providing, and integrating IT services, but there are some open issues by applying web services for telecommunication services. (Bo *et al.,* 2009)

However, web services can be used for the control and the management of value-added services in telecommunications and for integrating value-added services into IT processes. The integration of web services and value-added services is possible by use of the resource adaptor concept of JAIN SLEE that was described in section 4.4.

4.7 SIP Servlets

SIP Servlets are HTTP (Hypertext Transfer Protocol) Servlets that were extended with a Java programming interface for SIP communication and run on an SIP Application Servers (ASs). SIP Servlets are standardised as SIP Servlet API in JSR 116 (Java Specification Requests) (JSR 116, 2003) and JSR 289 (JSR 289, 2008). The AS provides the Servlet Container for the SIP Servlets. The Servlet Container is the Java-based runtime environment for the SIP Servlets. An application router for the composition of different applications (in this case SIP Servlets) has also been standardised.

The AS contains, among others, the SIP protocol stack. Received SIP messages are filtered according to the content of a specific configuration file, the deployment descriptor. Subsequently, the SIP Servlets, which correspond to the desired service, are executed. Figure 4.13 presents the structure of an SIP Application Server with SIP Servlets (IETF, 2001b).

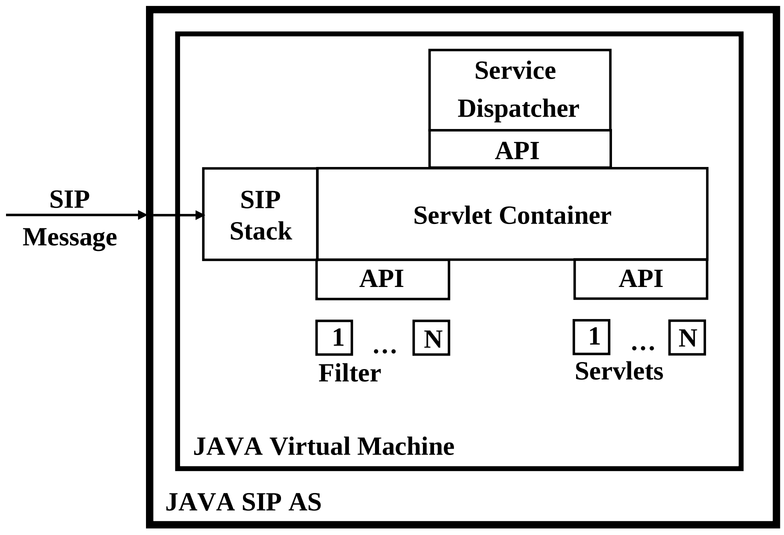


Figure 4.13: Application server and SIP servlets (related to (Trick and Weber, 2009))

The SIP Servlet API is standardised by the Java Community (JSR 116, 2003), (JSR 289, 2008). In this standard, the interworking between HTTP Servlets and SIP Servlets is defined. For example, in Figure 4.14 a converged service is shown, which consists of the SIP Servlet B and a HTTP Servlet C. Both servlets are part of one service that supports two protocols. The SIP Servlet B communicates with a SIP user agent and the HTTP Servlet C interacts with an HTTP client. The AS provides both, a SIP Servlet Container and an HTTP Servlet Container.

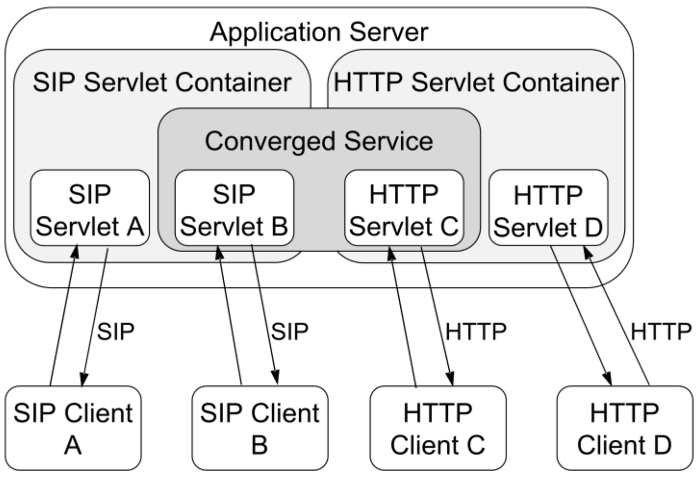


Figure 4.14: AS with SIP and HTTP servlet container

Unlike the SIP-CGI approach in Figure 4.11, the SIP Servlet Container replaces the CGI script in (Figure 4.15). The SIP Servlet Engine is invoked instead of the CGI scripts. The SIP Servlet Engine calls the corresponding Java method on the SIP Servlet. (Fan *et al.,* 2006)

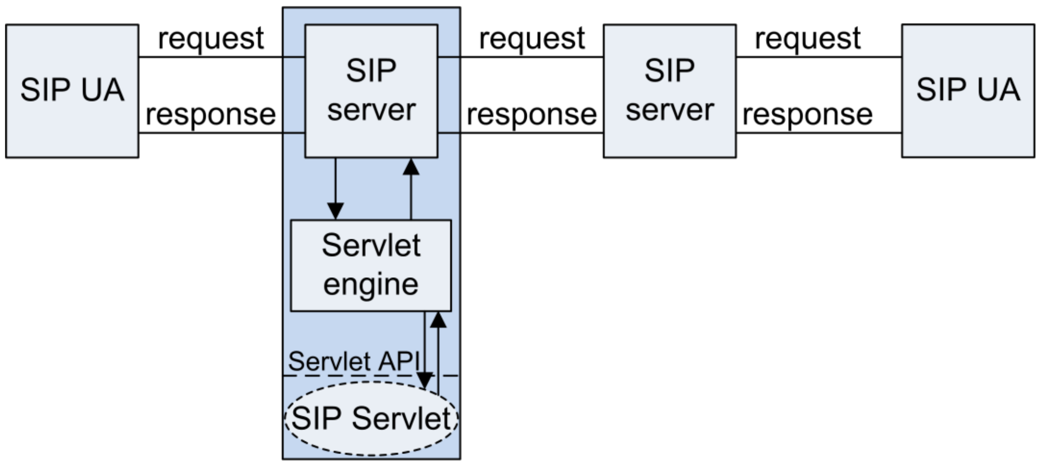


Figure 4.15: Servlet model for SIP (related to (Fan *et al.,* 2006))

SIP Servlets are persistent, and as they run in threads and not in processes like most CGI implementations, they have a higher execution speed. Servlets also provide a high level of security because they run within the SIP server process. Therefore, they are only accessible through the server itself. Servlets also offer all the benefits of

Java technology, such as platform independence and extensibility, and many service possibilities.

The disadvantage of the SIP Servlet technology is their exclusive support for SIP. To enable the usage of other protocols, converged applications between SIP servlets and Java EE technologies are required. Furthermore, the implementations of the protocols are vendor-specific, unlike the JAIN SLEE technology with its standardised RAs. Since servlets were originally defined for the HTTP protocol, they follow the client/server principles and cannot send initial requests messages. However, this problem can also be solved by using Java EE technologies in combination with SIP Servlets. Nevertheless, the JAIN SLEE framework already offers a solution, which offers the requirements for service execution, and provisioning, therefore it is not required to build another framework on top of Java EE in combination with SIP Servlets.

4.8 Related Research Projects

This section describes related research projects in the field of service execution and provisioning. Most of the analysed related research projects offer both, a service execution and a service creation solution. This chapter concentrates on the solutions for service execution and provisioning, nevertheless, if a related research project provides also a service creation solution this will be mentioned here.

MAMS

The objective of the BMBF-Project MAMS (Multi-Access, Modular-Services Framework) (MAMS, 2010) was to specify and rollout a novel, unified, open Service Delivery Platform for Next Generation Networks (NGN) and Services. The developed Service Delivery Platform (Figure 4.16) enables the rapid design of new combinable services for a wide range of multimedia applications based on the use of various network technologies and integrated voice and data. The service generation uses a collection of core communication services that are based on the interfaces of the OMA SE standard as described in section 4.3.

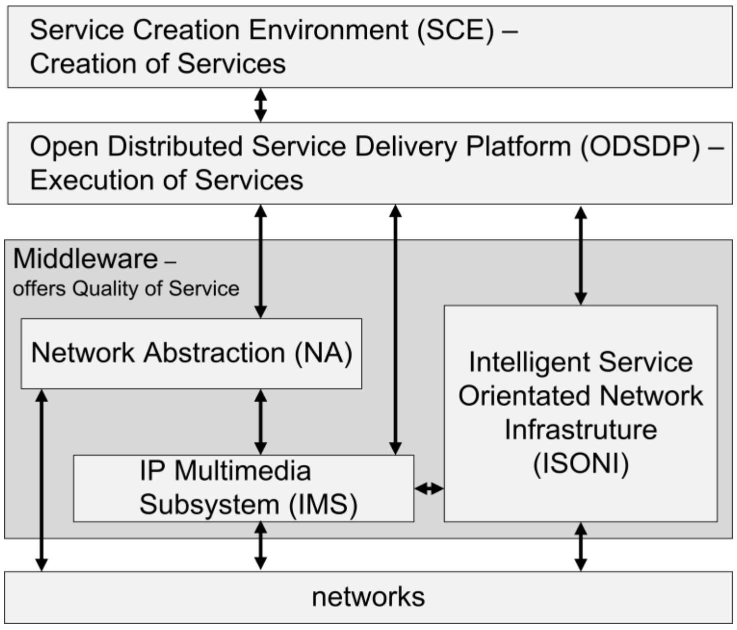

Service Creation Environment (SCE) – Creation of Services
Open Distributed Service Delivery Platform (ODSDP) – Execution of Services
Middleware – offers Quality of Service
Network Abstraction (NA)
Intelligent Service Orientated Network Infrastruture (ISONI)
IP Multimedia Subsystem (IMS)
networks

Figure 4.16: The MAMS framework (related to (Fraunhofer SIT, 2014))

The MAMS framework consists of an SCE which support the service creation process, an SEE that is called "Open Distributed Service Delivery Platform" (ODSDP) for the provisioning and execution of the services, and a middleware which consists of the IMS, a network abstraction (NA) and the Intelligent Service Orientated Network Infrastructure (ISONI). The ODSDP and the IMS provide the overlay to ISONI, which is based on reconfigurable, programmable nodes. The ISONI offers, among other things, reliability and quality of service. (Fraunhofer SIT, 2014)

With the help of a Service Creation Workbench (Freese et al., 2007) of the MAMS-Project, new value-added services can be created also by non-experts. The graphical user interface allows creating data flow oriented services. A service consists of preconfigured atomic services that are no more decomposable or interruptible.

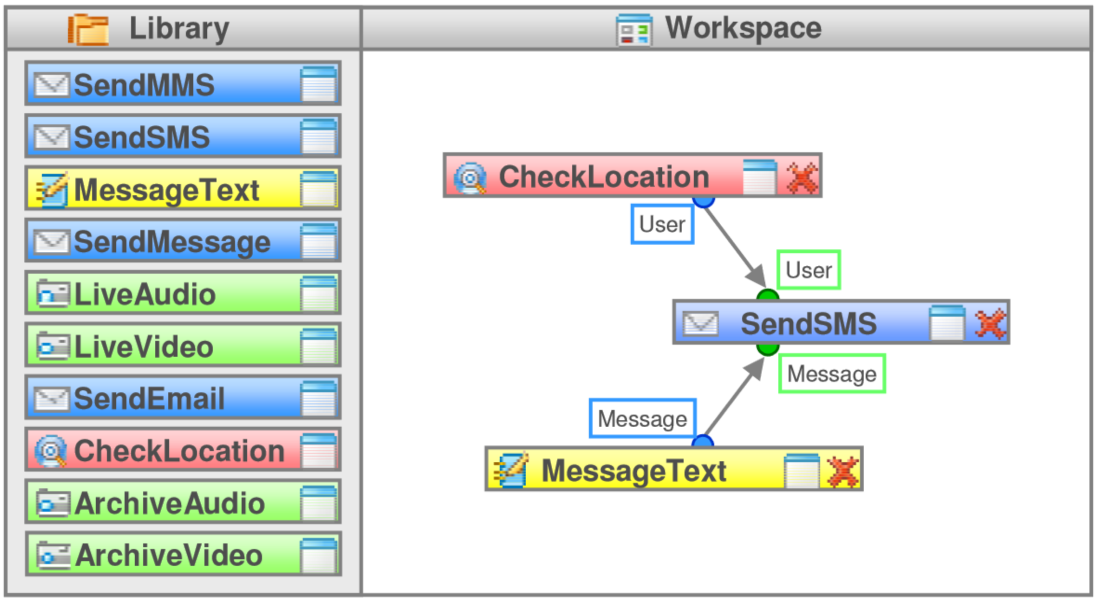


Figure 4.17: MAMS Service Creation Workbench (Freese et al., 2007)

Since MAMS is based on the OMA SE interfaces, it shares its inherent advantages (expandability and performance) and disadvantages (limited service possibilities and limited collection of core communication services). The MAMS framework consists of a proprietary Service Creation Workbench. Neither the OMA SE (refer to section 4.3) solution for the SEE nor the solution for the SCE is a solution for the framework proposed in this thesis (refer to section 2.5).

Orchestrated Execution Environment for Hybrid Services

(Bessler *et al.*, 2007) proposes an approach, using a BPEL (refer to section 3.6) engine deployed within a JAIN SLEE RA (Figure 4.18). The internal communication with the BPEL engine is implemented in Java and the communication with external web services uses SOAP. The BPEL resource adapter allows defining BPEL processes, which combine telecommunication services and web services.

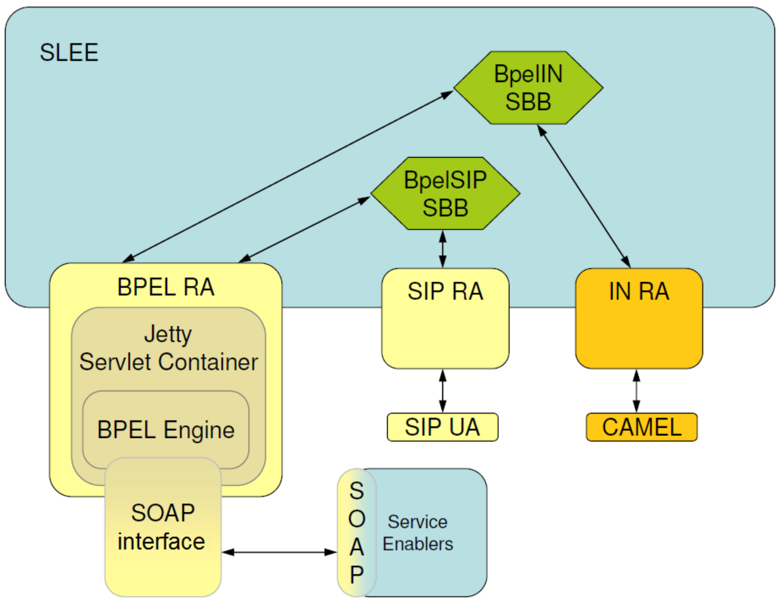


Figure 4.18: High-level architecture of the converged execution environment (Bessler *et al.*, 2007)

This approach inherits the advantages and disadvantages of web services and JAIN SLEE. Web services and JAIN SLEE services can be combined within one BPEL process. The technology is expandable because it is based on Java and open for many services. By adding new resource adaptors, new protocols can be supported.

The disadvantage of this approach is the lower performance compared to JAIN SLEE, which is caused by using a BPEL Engine for service orchestration.

Orchestrated Execution Environment Based on JBI

In (Bo *et al.,* 2009), the authors propose using BPEL (refer to section 3.6) for service orchestration. To overcome the performance limitations of Java EE-like application servers, this solution uses the JAIN Service Logic Execution Environment (JAIN SLEE) that, due to its event-based service platform architecture, is suitable for telecommunication services. An overview of the proposed architecture is shown in Figure 4.19.

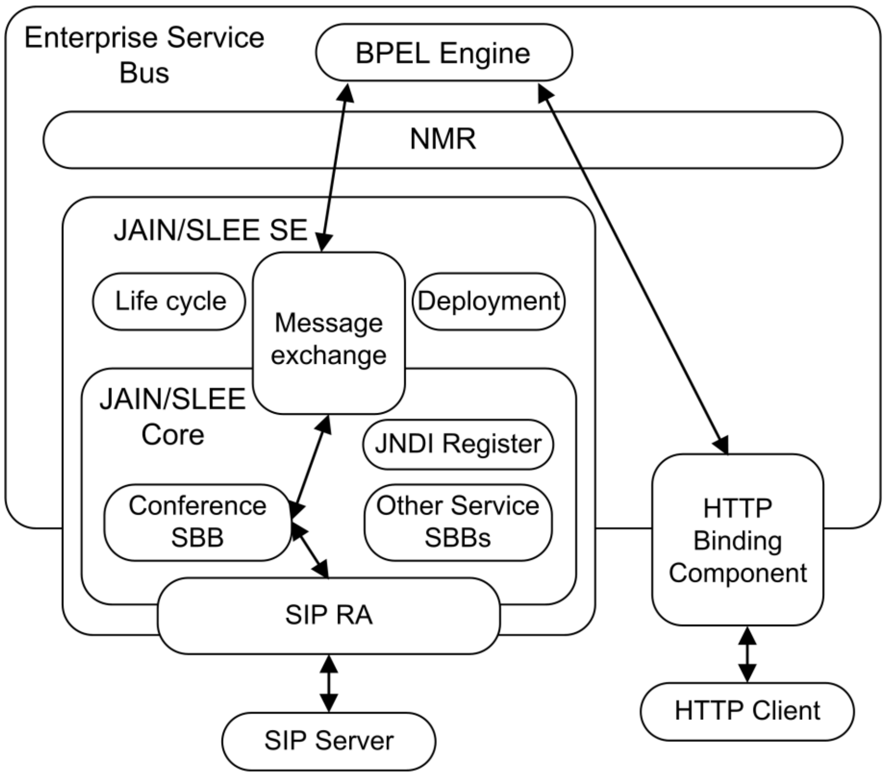


Figure 4.19: Architecture of the ServiceMix; Mobicents integration (Bo *et al.,* 2009) © 2009 IEEE

The figure shows the proposed architecture that consists of the BPEL engine for orchestrating web services, the Enterprise Service Bus with the NMR for exchanging messages between the components, the HTTP Binding Component for communicating with the web clients, and the JAIN SLEE Service Engine (SE) for providing telecommunication services. The JAIN SLEE SE consists of the integrated JAIN SLEE server, a life cycle module, a deployment module, and a message exchange module. The components communicate with each other by exchanging Normalized Messages (NM) via a router, the Normalized Message Router (NMR).

The life cycle of the modules is scheduled by the life cycle element. The message exchange module is a bridge between the NMR and the JAIN SLEE server. It receives NMs from the NMR, generates JAIN SLEE events from the NMs, and fires these events to the corresponding activities in the SLEE. The deployment module monitors a specific folder in the file system. It scans this folder for new DUs (Deployable Units). If a new DU is found within this folder, it is deployed to JAIN SLEE.

The JNDI module adapts the JNDI (Java Naming and Directory Index) APIs from JAIN SLEE to the ESB. A JAIN SLEE service requires a web service implementation with the related WSDL file to be available for service orchestration in BPEL.

For the interaction between the web service and a JAIN SLEE service, a SOAP RA is used which is acting as a communication bridge. The BPEL engine communicates via the NMR by sending Normalized Messages (NM).

Furthermore, a graphical Service Creation Environment is proposed that can expose and re-use telecommunication web services. An IT-developer can use the exposed WSDL interfaces for creating value-added services and communication web services without knowing the underlying technical details of the telecommunication protocols.

The example implemented by the authors is a conference service. It consists of the BPEL process and several SBBs. For mixing the audio/video streams, an external Media Server with a mixer is used. To make a user join a conference, a web client sends the request to the HTTP module. This module forwards the generated NM via the NMR to the destination (BPEL engine). When a NM is received by the BPEL engine, the corresponding BPEL process is activated. This process also generates NMs for the communication with the JAIN SLEE services. These NMs are forwarded across the NMR to the message exchange of the JAIN SLEE SE. There, the events are generated and sent over the event router to the corresponding conference SBBs. These SBBs implement the logic of the services. They are also able to control the media server and to give orders whether to or not to send, receive, or mix the media.

A prototype of the system was implemented, this prototype consists of the BPEL engine ODE (Orchestration Director Engine) (ODE, 2013), the ESB implementation Apache ServiceMix (ServiceMix, 2013), and the JAIN SLEE implementation Mobicents with the required SBBs. In addition, the conference scenario was developed and deployed on the prototype. In the next step, the response times of the system were measured. ServiceMix with ODE and Tomcat providing atomic services was compared with ServiceMix with ODE and Mobicents providing atomic services. The Tomcat application server uses (synchronous) web services, and the Mobicents application server is a JAIN SLEE implementation (asynchronous). As result of their experiment, the authors showed that the prototype (ServiceMix, ODE, Mobicents) with Mobicents is up to 10 times faster than a version that uses Tomcat (ServiceMix, ODE, and Tomcat). With the presented approach, the authors offer an Orchestrated Execution Environment for one conference service.

However, a BPEL engine is used for the orchestration of the web services. As described in section 4.6, this is not a solution for this work. Furthermore, the JAIN SLEE services are not generated automatically. They have to be developed manually. These facts lead to the conclusion that this approach is not suitable for this work.

4.9 Conclusion

The previous sections described the common technologies used in telecommunications for service execution and service provisioning. Advantages and disadvantages of each technology were shown. Selected research projects using these technologies were shortly discussed.

In Table 4.1, the evaluation of the technologies with regard to the criteria of service execution and provisioning (refer to section 2.5) is briefly summarised.

In order to provide a flexible service execution environment, the technology should support multiple protocols as well as a mechanism to be able to add new protocols to the SEE. Because of this requirement, CSE, SIP CGI, and SIP Servlets are rather unsuitable choices.

The SIP CGI technology does not fulfil the criterion of a good performance. The orchestration of pre-built telecommunication services with web services is possible, but the orchestration of fine-grained components of telecommunication services to value-added telecommunication services will lead to performance problems. The same problem is true for the research projects Orchestrated Execution Environment for Hybrid Services (Bessler *et al.,* 2007) and Design of an Orchestrated Execution Environment based on JBI (Bo *et al.,* 2009), because they use a BPEL engine. CSE, OSA/Parlay, OMA SE, and the research project MAMS show limitations of service possibilities.

Table 4.1: Service execution and provisioning solutions

	Supported protocols	Performance of the framework	Service possibilities	Composition capability
CSE	for GSM	OK	limited	no
OSA/Parlay, Parlay X	Resource Adaptor	OK	limited	yes
OMA SE	Binding Interfaces	OK	limited	yes
JAIN SLEE	Resource Adaptor	OK	unlimited	yes
SIP CGI	SIP	slow	nearly unlimited	yes
SIP Servlets	SIP	OK	nearly unlimited	yes
Web services	independent (SOAP preferred)	slow	nearly unlimited	yes
MAMS	Binding Interfaces (from OMA SE)	OK	limited	yes
Orchestrated Execution Environment for Hybrid Services	Resource Adaptor from JAIN SLEE	OK for JAIN SLEE services; slow for web services and service orchestration with BPEL	unlimited	yes
Design of an Orchestrated Execution Environment based on JBI	Resource Adaptor (HTTP, SIP)	OK for JAIN SLEE services; slow for web services and service orchestration with BPEL	unlimited	yes

Almost all technologies mentioned here make use of reusable components and provide service composition possibilities. SIP Servlets and the Service Building Blocks (SBBs) within JAIN SLEE offer service composition. However, composition is limited to the service delivery platform of the operator itself. Web services allow service composition across platforms. Based on the identified advantages and limitations against the listed criteria, JAIN SLEE satisfies the requirements of service execution and is therefore the preferred execution environment (Eichelmann *et al.,* 2008).

As already described in section 4.4, JAIN SLEE is only a solution for the SEE; the criteria for service creation are not fulfilled (refer to section 2.5). The target of this thesis is the development of novel extensions of the JAIN SLEE framework to fulfil the criteria for service creation. Therefore, in the next chapter, novel approaches for service description, creation, and execution are analysed.

5 Novel Approaches for Service Description, Creation, and Execution

This chapter analyses novel approaches for the description, creation, and execution of value-added services. The analysis of existing technologies and related work in the previous chapters showed that the choice of BPEL and JAIN SLEE would be appropriate for service description and service execution, respectively. However, the output of a BPEL developer tool is the service description in form of XML-files, but the JAIN SLEE server requires a Deployable Unit (DU) with the compiled Java classes and the descriptor files as input. This chapter analyses methods to bridge the gap between these two technologies (Figure 5.1).

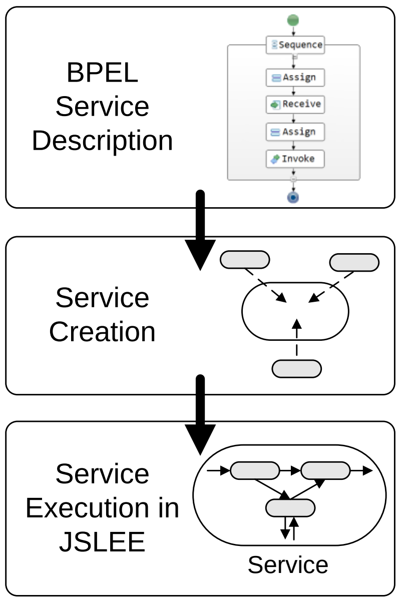


Figure 5.1: Service description, creation, execution

Section 5.1 defines the requirements for the proposed framework. Section 5.2 describes how BPEL is used for the service description. It is discussed how the service logic and the service functionalities are described in BPEL and which language elements are available for the service description.

The next section 5.3 addresses the creation of a new service. From the BPEL service description, a service has to be generated automatically. The result must be executable in JAIN SLEE. Alternative solutions for service creation are investigated and evaluated. The best approach is selected and based on the proposed approach the research framework is described in more detail in chapter 6.

In section 5.4, concepts for possible service structures are analysed and the best approach is selected and presented in chapter 7.

5.1 Requirements of the Proposed Framework

In chapters 3 and 4, the service creation and service execution technologies were evaluated. BPEL was selected as the service creation and JAIN SLEE as the service execution technology. To reach the goal of this thesis a framework, which supports an automated creation and provisioning of value-added telecommunication services, is required.

This section defines the requirements of the proposed framework, which are derived from section 2.5:

- An automated solution is required that supports the description, creation, execution, and provisioning of value-added telecommunication services.

- To support an easy and fast service development, the service description shall be supported by a graphical development tool.
- The developer needs to concentrate on building the logic of the service. Detailed knowledge of the communication protocols shall not be necessary.
- The developed framework shall support a broad range of value-added telecommunication services. Therefore, it shall be possible to describe a broad range of services in BPEL and, furthermore, the service execution environment shall support this broad range of services, too. Additionally, the framework shall support new functionalities and protocols.
- In order to provide the service designer with a simple and comfortable possibility to compose the service logic, reusable service building blocks have to be defined, which provide a mapping between the service description elements and the implemented logic elements in the SEE.
- Reusable service components shall offer the functionality for the value-added telecommunication services. They shall provide a mapping between the description of the functionality in the service description and the components implemented in the SEE. These CBBs shall support a coarse-grained functionality for a fast service development and a fine-grained functionality for a detailed service development. Furthermore, they shall support a wide range of communication protocols and the integration of new protocols.

Based on the defined requirements, different service description, creation, and execution approaches are researched in the following sections. From the results of this research, the framework is proposed in chapter 6 and the service structure of the value-added services is defined in chapter 7.

5.2 Service Description Concepts

As result of the discussions in chapter 3, BPEL has been selected as service description language. This section shows how BPEL can be used to describe value-added services. First, the BPEL elements which could be used for describing the service logic are introduced. In a subsequent step, it is shown, how the support of communication protocols and other functionalities, such as database access and data processing can be integrated into the service description.

5.2.1 BPEL for Service Description

The intention of the framework is to generate services in a simple and fast way. Therefore, the framework requires a description language that is simple but powerful enough to describe telecommunication services.

As already shown in section 3.6, BPEL (OASIS, 2007) is a description language that fulfils these requirements.

In contrast to the other solutions discussed in chapter 3, no BPEL engine is used in the approach proposed for this thesis, BPEL is only used as service description language. Therefore, BPEL processes, generated with the service description tool need not necessarily be BPEL processes, which are executable in a BPEL engine.

The service developer can use any BPEL description tools of his choice, to describe the service. If a standard BPEL description tool is used, the output will be an XML (Extensible Mark-up Language) (W3C, 2008) file. It is also possible for the developer to use a simple text or XML editor to describe the BPEL process. Another possibility for creating service descriptions is the usage of an interactive web interface. This solution is described in chapter 6.

WSDL files are used for the description of the partner links in the BPEL process. With these partner links, the functionalities can be selected from the CBBs and combined with the service logic (refer to section 5.2.3). For complex transformations and expressions including loops, if statements, and other conditions within the BPEL process, XPath (XML Path Language) (W3C, 1999) can be used.

BPEL processes are normally deployed on a BPEL engine and are executed as web services. In the proposed framework, BPEL processes neither are deployed on a BPEL engine nor are developed as web services. Here, the output of the BPEL development tool is passed to the Code Generator (refer to section 5.3.1) or to the Service Description Parser of the Runtime Service Composition concept (refer to section 5.3.2).

5.2.2 Describing the Service Logic in BPEL

The service can be described in a graphical way or by editing a XML document. A typical BPEL editor is the Eclipse BPEL Designer (Eclipse, 2013). This tool supports both the GUI-based and the XML editor-based process development. In Figure 5.2, an example of the GUI-based BPEL editor is shown.

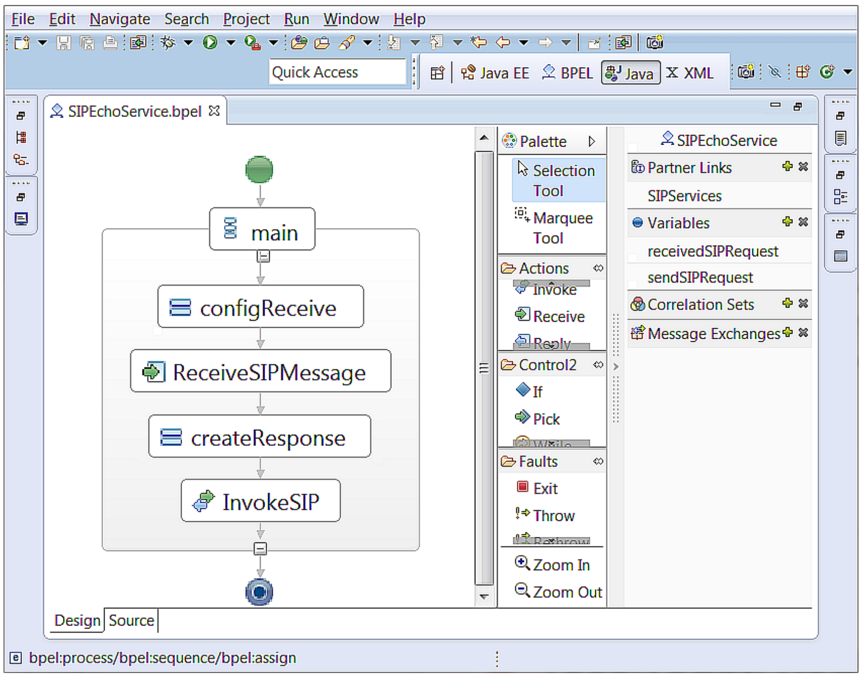
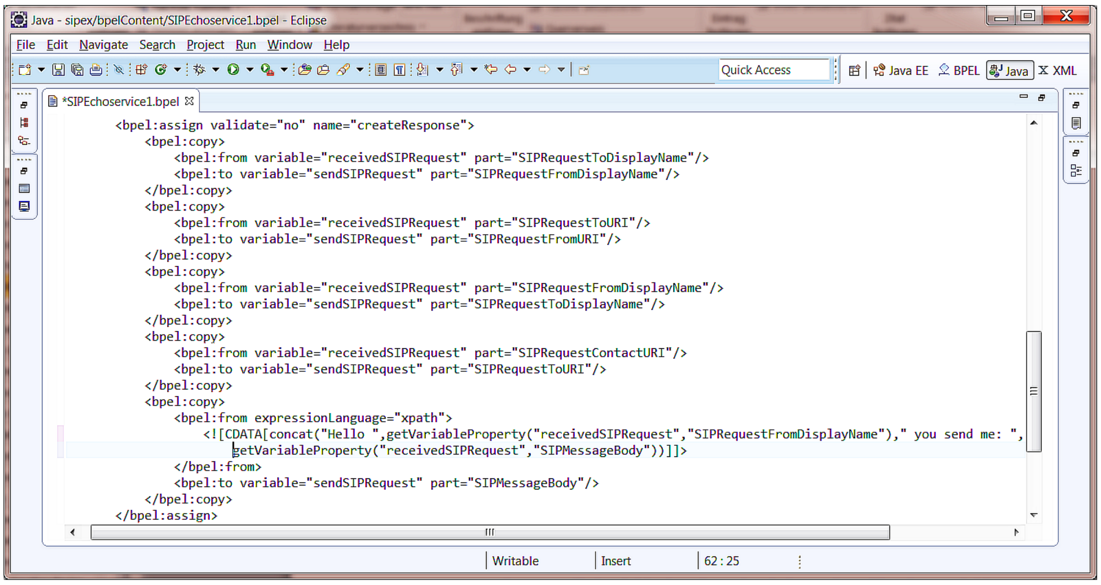


Figure 5.2: Eclipse BPEL designer

The service developer can build the BPEL process by dropping process elements, called “activities”, into the main window and combine these activities with arrows to form a graphical representation of a state machine. The graphical process representation has a start point and an end point. Between these points, the activities can be placed. The service execution will begin at the start point and finish at the end point. The arrows will mark the execution direction. Alternatively or in combination with the GUI editor, the XML document can be edited directly. For example, the XML representation of the “createResponse” activity from the BPEL process displayed in Figure 5.2 is shown in Figure 5.3.


Java - sipex/bpelContent/SIPEchoservice1.bpel - Eclipse
File Edit Navigate Search Project Run Window Help
Quick Access
Java EE
BPEL
Java
XML
*SIPEchoservice1.bpel
Writable
Insert
62 : 25

```
<bpel:assign validate="no" name="createResponse">
    <bpel:copy>
        <bpel:from variable="receivedSIPRequest" part="SIPRequestToDisplayName"/>
        <bpel:to variable="sendSIPRequest" part="SIPRequestFromDisplayName"/>
    </bpel:copy>
    <bpel:copy>
        <bpel:from variable="receivedSIPRequest" part="SIPRequestToURI"/>
        <bpel:to variable="sendSIPRequest" part="SIPRequestFromURI"/>
    </bpel:copy>
    <bpel:copy>
        <bpel:from variable="receivedSIPRequest" part="SIPRequestFromDisplayName"/>
        <bpel:to variable="sendSIPRequest" part="SIPRequestToDisplayName"/>
    </bpel:copy>
    <bpel:copy>
        <bpel:from variable="receivedSIPRequest" part="SIPRequestContactURI"/>
        <bpel:to variable="sendSIPRequest" part="SIPRequestToURI"/>
    </bpel:copy>
    <bpel:copy>
        <bpel:from expressionLanguage="xpath">
            <![CDATA[concat("Hello ",getVariableProperty("receivedSIPRequest","SIPRequestFromDisplayName")," you send me: ",
                getVariableProperty("receivedSIPRequest","SIPMessageBody"))]]>
        </bpel:from>
        <bpel:to variable="sendSIPRequest" part="SIPMessageBody"/>
    </bpel:copy>
</bpel:assign>
```

Figure 5.3: XML-Editor from Eclipse BPEL designer

The service logic is described using BPEL activities. A list of the activities defined in BPEL (OASIS, 2007) is given in Table 5.1.

Table 5.1: BPEL activities (OASIS, 2007)

Activity	**Description**
Invoke	The Invoke activity is used to describe a call of a method defined within a partner link. The methods represent the available functionality that can be used within the service.
Receive	The Receive activity is used to describe a point in the workflow of the service where the workflow should wait for an incoming event. This activity is used to describe the possibility that the service can be called from other services and resources, or it waits for replies from other services and resources.
Reply	The Reply activity describes the possibility that the service can send a reply to an event that was received. The combination of a Receive activity and a Reply activity can form a request-response operation for the service.
Assign	The Assign activity can be used to copy data from one variable to another, insert literals into variables, and insert new values into the variables by using expressions.
Throw	The Throw activity is used to define when a service instance needs to signal an internal fault.
Wait	The Wait activity is used to define a deadline or a delay for a period. The Wait activity will end when the specified deadline or duration has been reached.

Empty	The Empty activity does nothing; it can be used, e.g., for suppressing a fault that needs to be caught, or for providing a point of synchronization in a Flow activity.
Extension	The Extension activity is used to define new activities that are not defined in this table. This is a BPEL-conform possibility to add new individual activities to the framework.
Exit	To end the service execution immediately, the Exit activity can be used.
Rethrow	The Rethrow activity is used to propagate faults. It is applied in fault handlers.
Sequence	A Sequence activity is a container for one or more activities that are executed sequentially, i.e. in the lexical order in which they appear within the service description of the Sequence activity.
If	Conditional behaviour can be described with the If activity. The If activity contains a list of one or more conditional branches. This branches are the required “if” branch, the optional “elseif” branch and the “else” branch. The order in the list of branches also corresponds to the order in which the conditions are analysed. If a condition is true, then the corresponding branch will be executed; if this condition is false, the next condition will be analysed. If no condition is true, the “else” branch will be executed. The If activity will end, when the contained activities of the selected branch have ended, or will end immediately, when no condition is true and no “else” branch has been specified.
While	The While activity offers a mechanism for a repeated execution of the contained activities. A Boolean condition is used to check whether the contained activities are executed or not. The condition is analysed for all iterations. Only if the condition evaluates to true, the contained activities will be executed.
RepeatUntil	The RepeatUntil activity offers a mechanism for repeated execution of the contained activities. A Boolean condition is used to check whether the contained activities are executed or not. The condition is analysed after each execution of the loop. Only if the condition evaluates to true, the contained activities will be executed. In contrast to the “While” loop, the “RepeatUntil” loop executes the contained activities at least once.
Pick	The Pick activity describes the possibility to wait for one event from a set of events. It can receive different events and will wait, until one of the events have been received; then it will execute the activity associated with that event. After an event has been received, no other event will be accepted by the Pick activity. The Pick activity will have ended after the selected activity has finished.

Flow	The Flow activity provides the developer with the possibility to describe parallel executions. This activity can consist of multiple branches. These branches can be executed in parallel. Each branch can include further activities. The flow activity will end, after all branches with their activities have been executed.
ForEach	The ForEach activity represents a loop that executes the contained activity for a specified number of times. The ForEach activity can execute the contained activity in a parallel or sequential order.
Scope	The Scope activity is used to define a nested context. A Scope requires a subordinate activity that can be a complex activity which contains further activities. The provided context is shared for the nested activities.
Compensate	The Compensate activity is used to support compensation for inner Scopes. It compensates all inner Scopes that have already completed successfully.
CompensateScope	To compensate a Scope activity that has already ended successfully, the CompensateScope activity is used.
Validate	The Validate activity is used to validate the values of variables against their associated data definition.

These activities consist of logic elements required to describe a service. With the activities “invoke”, “receive”, and “reply” it is possible to describe a waiting state for an event from the partner or to call methods at the partner. This mechanism can be used to integrate resources and functionalities into the service.

5.2.3 Describing the Functionality in BPEL

The previous section showed how the service logic could be described with BPEL. This section puts the focus on the description of functionalities which can be integrated into the service. The functionalities can be provided by the service itself, e.g., mathematical calculations or they can be provided by external applications, e.g., mixing of video streams, audio encoding, communication with smart devices and home automation devices. The functionalities are implemented in methods within the

SEE. These methods provide, e.g., the functionalities which handle the protocol communication with external applications.

The normal way to add functionality to a BPEL process is to use the BPEL partner links to call external web services. In BPEL, the partner links are described with the Web Service Description Language (WSDL) (W3C, 2007b). WSDL is a platform and protocol-independent programming language for defining the interfaces of web services. WSDL is a meta-language that allows the description of the offered functionalities, data types, and data exchange protocols of a web service. The operations that are accessible from the outside, as well as the parameters and return values of these operations are defined in the WSDL files.

The proposed framework does not make use of a BPEL engine to orchestrate the web services. Instead, all services generated from the BPEL description run on a JAIN SLEE server (Figure 5.1). With a web service resource adaptor, the JAIN SLEE service can use also web services, but, as already said in chapter 3, this is not a solution for the proposed framework.

The idea is to utilise the partner links only for a description of required functionalities. Then at the time of service creation, this description is mapped to the corresponding service components of the SEE.

With this approach, it is possible to use BPEL for the description of the required functions and the developer can also make use of standard BPEL developer tools to add the required functionality through partner links.

For the mapping from the BPEL description of the functionality to the JAIN SLEE component implementing the functionality, new Communication Building Blocks

(CBBs) (Eichelmann *et al.*, 2008) (refer to section 6.2.1) are defined. Every partner link can be represented by one CBB. To describe the invocation of a functionality in BPEL, the correspondent method of the partner link has to be invoked in the BPEL process.

The functionalities which are described through the partner links are not necessarily complete services. Also very fine-grained functionalities can be described as partner links; e.g., a mathematical function or a string parse operation. With this possibility, the level of abstraction can easily be customized. The range spans from a high-level of abstraction with the definition of coarse-grained partner links to a low level of abstraction with the definition of fine-grained partner links. Examples of different level of abstractions are presented in Figure 5.4, Figure 5.5, and Figure 5.6.

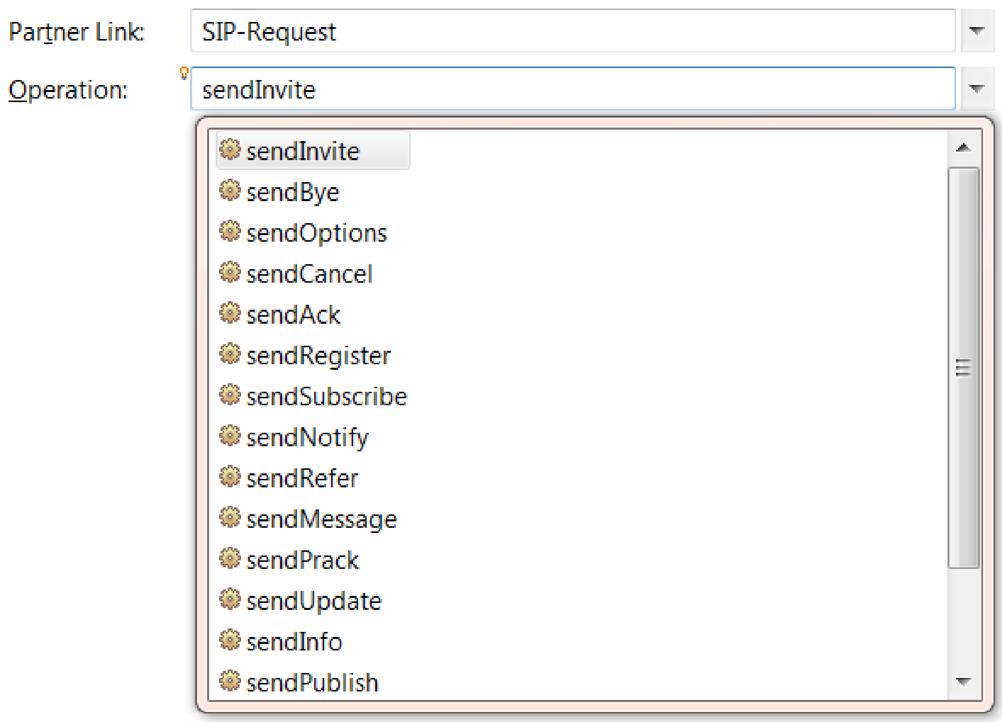
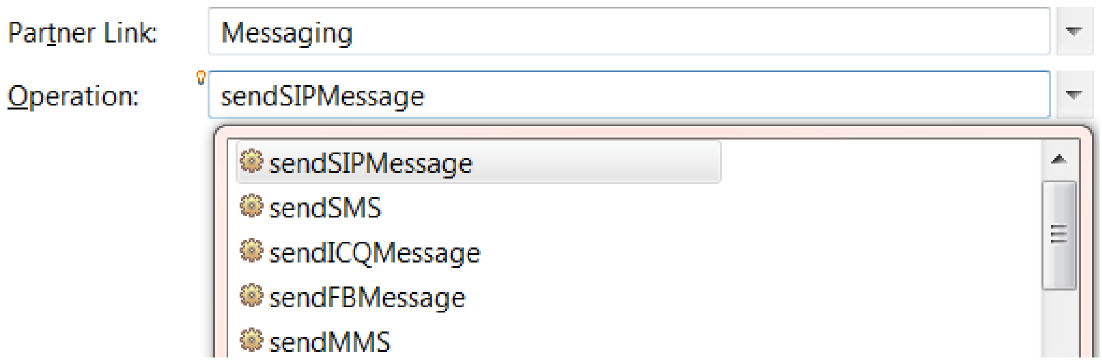

Partner Link: SIP-Request
Operation: sendInvite

Figure 5.4: Choosing a functionality from a BPEL partner link (protocol level example)

In Figure 5.4 the partner link “SIP-Request” offers some protocol-specific functionality. A drop-down list offers the available operations of the partner link. In this case, the operation “`sendInvite`” is selected from the list of possible operations, which allows to configure a SIP Invite request.

The next example (Figure 5.5) shows a partner link called “Messaging” which allows to send out an instant message. This partner link offers functionality with a medium level of abstraction. The level of abstraction is higher than in the first example, but lower than that of the third example.

In this example, the service developer has to define the operation to send out an instant message. Here, the “`sendSIPMessage`” operation is selected from the drop-down list. Therefore, the service will use the SIP protocol to send out instant messages. The service developer can choose the protocol for the instant message, but does not need to take care about the protocol-specific communication.



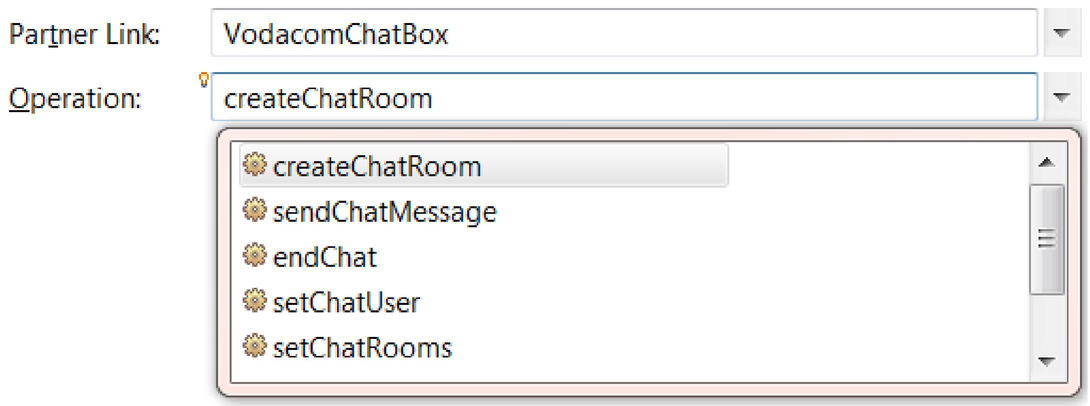
Partner Link: Messaging
Operation: sendSIPMessage

- sendSIPMessage
- sendSMS
- sendICQMessage
- sendFBMessage
- sendMMS

Figure 5.5: Choosing a functionality from a BPEL partner link (medium level example)

The third example (Figure 5.6) presents a chat room functionality. The developer can easily integrate the complete chat functionality into the service by using a very high abstraction level. Here, the complete chat communication is handled by the implementation and cannot be manipulated by the service developer. In this example, the partner link is called "VodacomChatBox". This partner link offers operations to create and control a chat room. Here, the operation "`createChatRoom`" is selected from the drop-down list, which allows the user to configure a chat room.

The partner link "VodacomChatBox" can also be a third-party application. The third-party developer has to build the components that implement the functionality and the description file of the partner link.



Partner Link: VodacomChatBox

Operation: createChatRoom

Figure 5.6: Choosing a functionality from a BPEL partner link (high-level example)

The partner links allow the service developer to describe the functionality of a service in the same way as in the case of external web services are orchestrated. New functionalities that are implemented within the JAIN SLEE components can simply be added to BPEL by describing new partner links. The CBBs support the mapping from the partner links in BPEL to the implementation within the proposed framework (refer to chapter 6).

5.3 Service Creation Concepts

Once the service has been described with BPEL, the service description needs to be translated into an executable implementation of the value-added service. The proposed framework does not use a BPEL engine in combination with web services.

The value-added service implementation runs on a JAIN SLEE server (refer to section 4.4). Therefore, a new solution is required to generate JAIN SLEE-based value-added services from the BPEL service description. This work introduces two new research approaches. The first approach is called the “Code Generator approach”. This technique creates the service as a Deployable Unit (DU) that is deployable on the JAIN SLEE application server (refer to section 5.3.1). The second approach, Runtime Composition, composes the service from pre-defined service components directly within the JAIN SLEE service container (refer to section 5.3.2). Both approaches are evaluated, and the best approach, the second approach, is chosen for the proposed research framework (refer to chapter 6).

5.3.1 Code Generator

The Code Generator (Eichelmann *et al.,* 2008) will transform the BPEL process files into Java source code and descriptor files, and create a Deployable Unit (DU). A DU is a packed folder that includes the source code files, the descriptor files, and all resources that are required to run the service on the application server. An overview of the Code Generator is given in Figure 5.7.

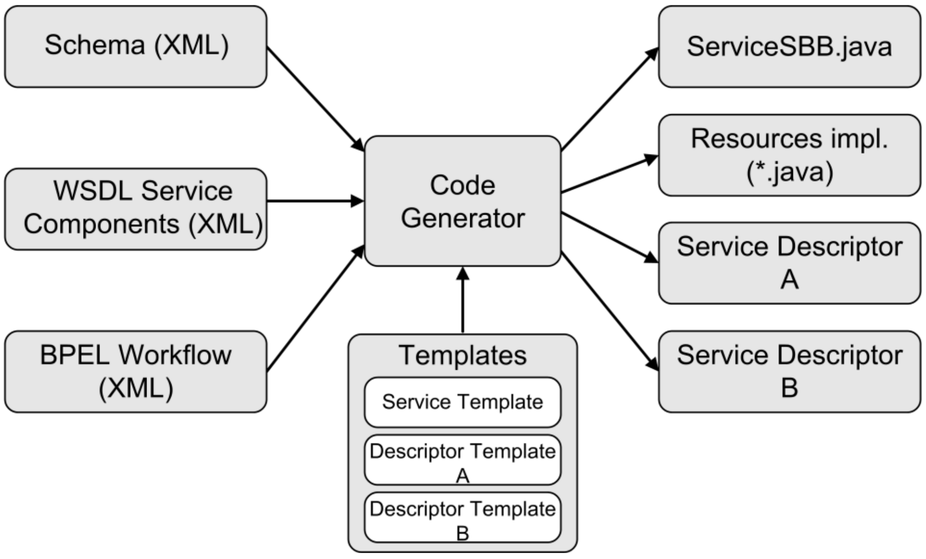


Figure 5.7: Code generator

As input, the Code Generator uses the BPEL process file “BPEL Workflow”, the WSDL files created by the BPEL developer tool “WSDL Service Components”, and the XML schema files “Schema”. The input files are systematically analysed. For each BPEL element which is found in the descriptions, such as activities, variables, or methods that are called from a partner link in BPEL, the Code Generator will add pre-defined code snippets to the Java “Service Template” and XML snippets to the “Descriptor Templates”. From these template files, the Java and descriptor files are generated. In the example which is given in Figure 5.7, the Code Generator creates the two descriptor files “Service Descriptor A” and “Service Descriptor B”, i.e. a Java file “ServiceSBB.java” that contains the source code of the service, and a Java file that implements the functionality described in the partner links.

Generally, the descriptor files define the components of the JAIN SLEE service, such as the SBBs, the events, properties, and the RAs (refer to section 4.4). The SBBs that are defined in the descriptor files are implemented as Java classes. Furthermore, the partner link methods that are used in BPEL are also implemented as Java classes that can be called from the SBBs. They are mapped to one or more resource adaptors or to other functionalities and resources (refer to section 5.2.3). The BPEL activities, the service workflow, and the variables are also mapped to Java code snippets. For each element used in a BPEL process, a pre-defined Java code snipped is required.

When the parser reads the BPEL process for each element that was found, the associated Java snippets with the defined parameters are added to the source templates of the SBBs. For every new resource adaptor that should be supported by the Code Generator, Java snippets and Java classes have to be defined. The Java classes and the code snippets implement the functionalities which are required for using the new protocol that is supported by the RA. These code snippets contain the method calls of the Java classes that implement the functionality. The results of the code generation process are the Java classes and the descriptor files that have been generated from the templates. From these files, the DU is created.

To be able to call the available functionalities (refer to section 5.2.3), they need to be available in the BPEL service description. As shown in the last section, resources and functionalities are described with the help of partner links in BPEL. Methods and attributes which require the functionalities must be added as code snippets to the code generator and have to be described within the WSDL file of the responsible partner link.

When the Code Generator generates a new service, it will create a new workspace with the needed Java code, descriptor files, build files, and libraries. After the creation of the workspace and the Java files, the Code Generator creates the Deployable Unit (DU) that includes all generated files. With the help of an Apache

Ant (Ant, 2010) script, this deployable unit can be deployed on a JAIN SLEE Application Server. The Ant script copies the DU to the application server and extracts it. Finally the service can be executed.

5.3.2 Runtime Service Composition

With the Runtime Service Composition approach (Eichelmann *et al.,* 2011) the services will be generated on start time from pre-deployed service components (Figure 5.8).

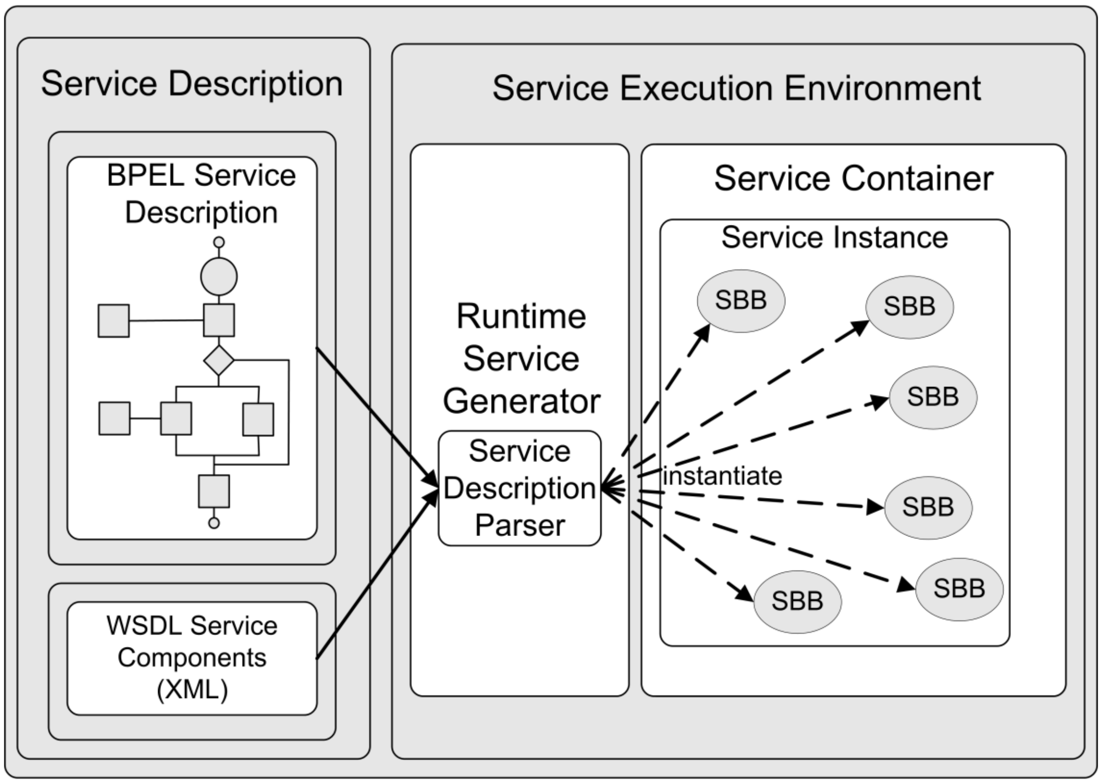

Service Description
BPEL Service Description
WSDL Service Components (XML)
Service Execution Environment
Runtime Service Generator
Service Description Parser
Service Container
Service Instance
SBB
SBB
SBB
instantiate
SBB
SBB
SBB

Figure 5.8: Runtime service composition

These service components offer the service logic and functionalities that are required for the service. Before a service can be composed, all of the required elements are deployed on the application server. To generate a service with the Runtime Service

Composition approach, the service description is uploaded to the framework, e.g., by a web interface, and passed to the service description parser. The service description consists of the developed BPEL process and WSDL documents.

The service description parser analyses the new service description, instantiates the required SBBs, initiates the service context (refer to section 7.1.1), and creates and initiates the required variables (refer to section 7.1.2) and service components (refer to section 7.1.5). The instantiated service is configured with the parameters from the BPEL service description. After the creation and configuration of the service instances, each service instance is triggered for execution.

The service logic and service functionalities are represented as SBBs, in contrast to the Code Generator approach code snippets are used. This fact offers good possibilities for third-party development. The third-party developers deliver the SBBs together with the partner link descriptions. To use the new functionality, the SBBs need to be deployed in the SEE and the partner links to be included into the new BPEL process. Therefore, it is very easy for the third-party developer to develop new functionalities, and it is easy for the BPEL developer to integrate these functionalities into the services.

This approach also offers the possibility of an easy monitoring of the service instances, for example, by requesting status events from the service components, and it is also possible to reconfigure the service at runtime, e.g., with a web-based service description and monitoring tool.

5.4 Service Execution Concepts

This section defines the representation of the service within the SEE and describes the service execution concepts. An extension of the JAIN SLEE framework is proposed as SEE (refer to section 6.3), so the service logic is mapped to components called “SBBs” (refer to section 4.4). A service can consist of one or more of these SBBs. Four concepts for service execution have been developed (Eichelmann *et al.,* 2009; Eichelmann *et al.,* 2010) during the research: (i) the “Single SBB concept” (refer to section 5.4.1), (ii) the “Parallel Program Flow concept” (refer to section 5.4.2), (iii) the “Orchestration concept” (refer to section 5.4.3), and (iv) the “Choreography concept” (refer to section 5.4.4).

5.4.1 Single SBB Concept

In this approach, the service logic is implemented in only one single SBB. Within this SBB, a state machine controls the service workflow. The state machine decides which events are allowed to be received by an individual state and the actions that are executed after an event has been received.

This concept can be used in combination with the Code Generator approach (refer to section 5.3.1). With this approach, the required functionality can be added to the templates before the SBB has been created. The Code Generator generates the state machine that represents the workflow of the BPEL process. This approach was analysed within the TeamCom project (Eichelmann *et al.,* 2009).

The Runtime Service Composition concept is not usable in combination with this approach, since generally logic functions and functionalities are represented as multiple service components whereas the "Single SBB concept" allows for one service component only.

An example of a service instance that consists of only one SBB is provided in Figure 5.9. The represented service is able to send and receive events from three RAs. It listens for incoming events from the RAs and can call methods from the RA interfaces.

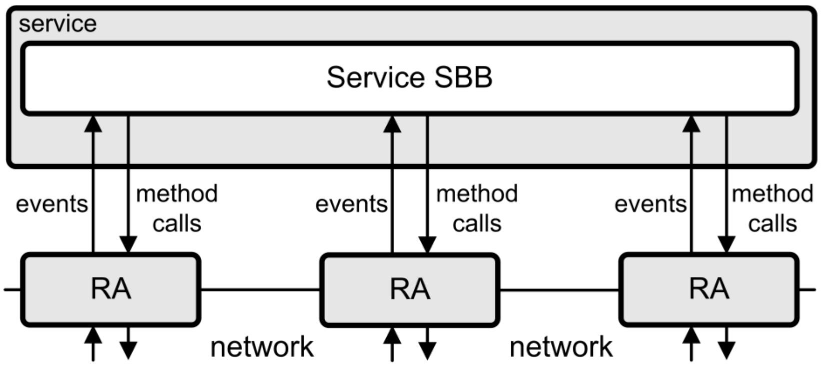


Figure 5.9: Single SBB concept

In Figure 5.10, a simple BPEL process is shown on the left side, which contains only three activities within its main sequence: a Receive activity called "receiveInput", an Assign activity called "assign", and an Invoke activity called "invokeCallback". This BPEL process contains a service description for an echo service, which receives incoming instant messages, and sends instant messages back to the sender.

Once the service has been started, it waits for an incoming event. This service logic is represented by the Receive activity of the process. In this case, it waits for an event that signals the reception of an instant message.

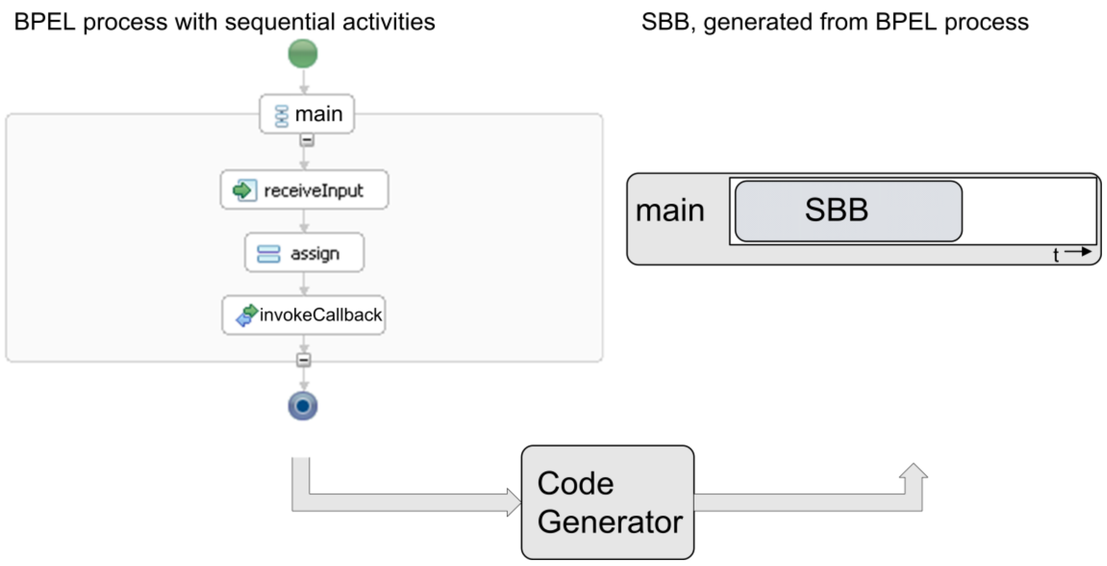

BPEL process with sequential activities
SBB, generated from BPEL process
main
receiveInput
assign
invokeCallback
main
SBB
t
Code Generator

Figure 5.10: BPEL process with sequential activities and the generated SBB

When a corresponding event is received by the service, the incoming message is analysed, and the sender address of the instant message together with the message body are stored in variables for a later usage in the answer message. In the BPEL process, this logic is represented as Assign activity.

In a next step, the new instant message that should be returned to the sender is prepared. The previously stored sender address is now used to address the new instant message, and the stored message body is used as new message body for this instant message. Then, the new instant message is sent back to the sender of the message. In the BPEL process, this behaviour is described with the Invoke activity “invokeCallback”.

With the Code Generator approach, each instance of a service consists of one monolithic SBB, and is not able to process multiple workflows in parallel.

The analysis of this approach has shown that it is sufficient for services that consist of a sequential workflow. Services that require parallel workflows are not supported by this approach. In JAIN SLEE, it is not allowed to use multi-threading within one SBB. Therefore, a new approach was required which is able to support parallel workflows.

5.4.2 Parallel Program Flow Concept

In order to realise parallel workflows, which is not possible with the "Single SBB concept", a new method is required.

A possibility to realise parallel program execution in JAIN SLEE is the usage of more than one SBB (Eichelmann *et al.,* 2009). The JAIN SLEE-compliant SEEs support multiple SBBs running in parallel, i.e. they send and receive events and perform multiple tasks at the same time. Parallel execution is required when several independent tasks have to be performed, e.g., forking or handling multiple connections in parallel, and for the composition of several service components.

In BPEL, the Flow activity and the ForEach activity can be used to describe the parallel program execution. If the Code Generator is reaching, for instance, the Flow activity while parsing a BPEL process, the Code Generator is generating a new BPEL process from each branch within the flow sequence.

Figure 5.11 shows the BPEL processes that are generated from the flow branches of a Flow activity within the main sequence of a parallel process.

This Flow activity contains two flow branches. A new BPEL process will be generated for each branch. Additionally, all new processes get a Receive activity as first activity of the BPEL process and an Invoke activity as last activity of the process. Later, from these two activities, the SBB Java methods will be generated which send and receive events for the communication with the main SBB. The newly generated BPEL processes are delivered to the Code Generator. As a result, the Code Generator can generate a new SBB for each BPEL process. When there are multiple flow activities, nested within the same BPEL process, this step is repeated until all flow branches will have been transformed into new BPEL processes and finally translated into SBBs.

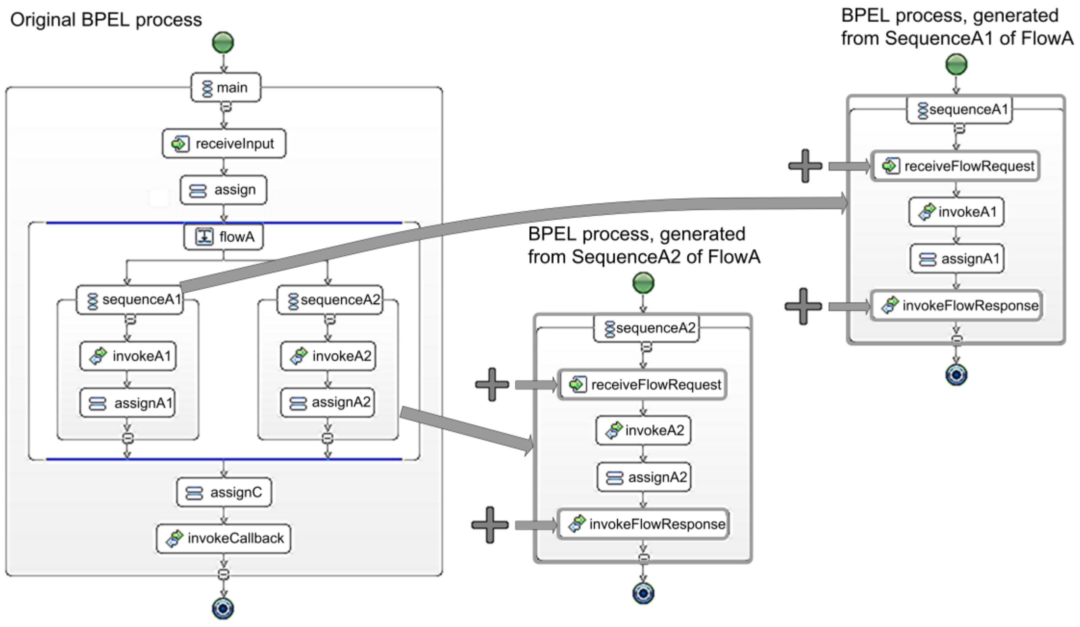


Figure 5.11: BPEL process with flow activity

In order to execute parallel service components in JAIN SLEE, the SLEE standard offers the possibility to use several SBBs within a service. These SBBs can be executed in parallel. Either each SBB can form its own service, or several SBBs can form a single service together. In this approach, several SBBs are used to represent the Flow activity. Each branch of the Flow activity is presented in its own SBB. One SBB represents all activities from one branch of the Flow activity. If a service consists of several Flow activities, it will be distinguished whether the Flow activities are contained in the same sequence or whether they are contained in another Flow activity.

The main SBB generated from the main sequence of the BPEL process uses request events to signal to the flow representation SBBs (flow SBBs) that are generated from the Flow activity in order to start processing. After the main SBB has fired these request events to all flow representation SBBs, it will wait for responses from the SBBs. With the request events, the flow SBBs will receive the required parameter values from the main SBB. These values are used to initialise and to activate the flow SBBs. After the processing of the flow SBBs, the parameter values are assigned to the response events and delivered back to the main SBB. Then, the main SBB receives the response events from all flow branches. The main SBB can copy the parameter values from the flow SBBs and will continue with the processing.

Figure 5.12 shows on the left side a BPEL process which has been extended by a Flow activity, and an Assign activity in comparison to the process in Figure 5.10.

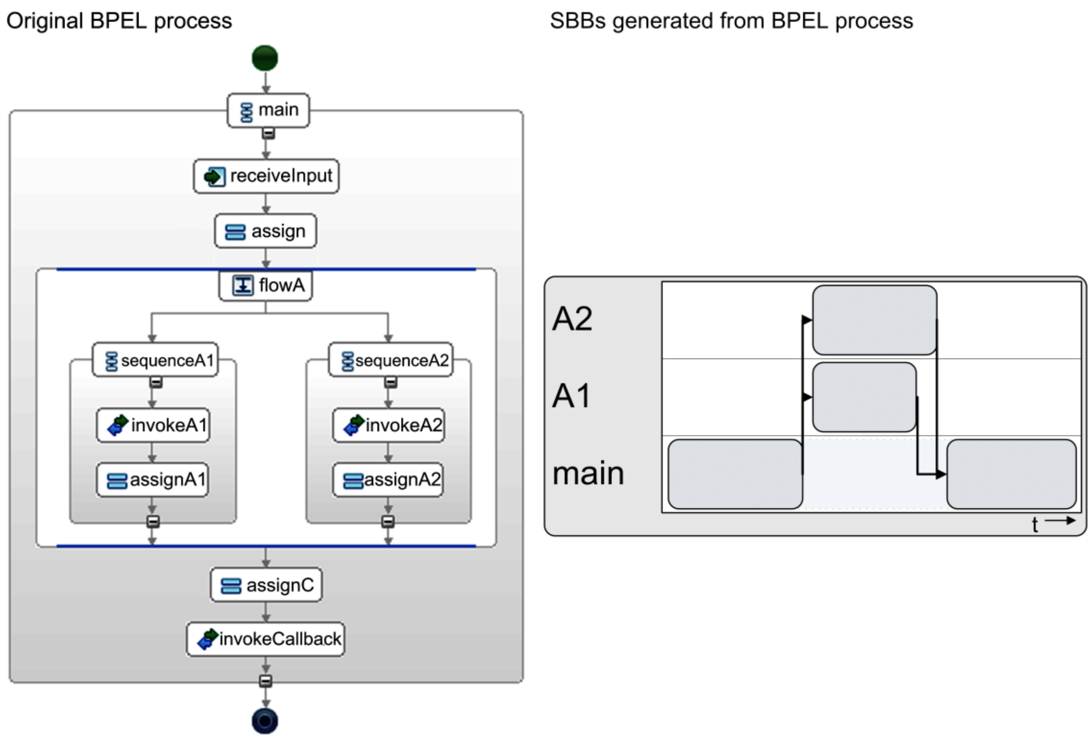


Figure 5.12: BPEL process with flow activity and the resulting SBBs

The additional Assign activity, which is called “assign”, is used to copy the variables, which have been returned by the flow SBBs into the variables of the answer message.

The sequences within the branches of the Flow activity are processed in parallel. If a sequence in a flow branch contains further sequential activities, the contained activities will be sequentially processed within the flow SBB, as in the case with the main sequence activities in the main SBB.

The BPEL process in Figure 5.12 contains two branches in its Flow activity. The Flow activity is called “flowA”, and the sequences are called “sequenceA1” and “sequenceA2”. In this example, both sequences contain two activities, an Invoke activity and an Assign activity. From the BPEL process, three SBBs are generated: the main SBB, SBB A1, and SBB A2. The main SBB of this service represents the sequential activities of the main sequence. The SBB A1 represents the activities from the first flow branch and the SBB A2 the activities from the second flow branch. The SBBs resulting from this BPEL process are shown on the right side of Figure 5.12.

Request and response events have been defined for the communication between the SBBs. Each SBB which has been generated from a flow branch is called with a request event and answers with a response event after its processing (Figure 5.13). Both the request and the response event contain the necessary variables that will be available in both SBBs. The generated SBBs and the direction of the request and response events are shown on the right side of Figure 5.12 and in Figure 5.13.

A BPEL process is not only limited to one Flow activity, it can also contain as many flow activities as desired on different nesting levels. Two different nesting possibilities must be considered. Several flows can be contained in the same sequence or flow activities can also be contained in the various branches of a flow.

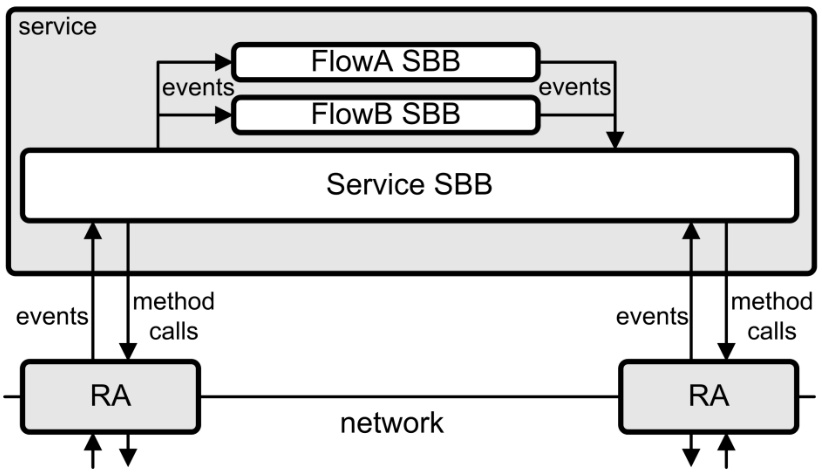


Figure 5.13: Parallel Program Flow concept

Figure 5.14 shows a simplified BPEL process that contains two flow activities in the same sequence. The first Flow activity is called “flowA”, and the second is called “flowB”. Each Flow activity consists of two flow branches, and in each branch a sequence with further BPEL activities is contained.

For each branch in each flow, a SBB is generated. Each SBB represents the activities of the appropriate branch in Java code. For instance, the first branch of the “flowA” activity with the sequence “sequenceA1” from the BPEL process shown in Figure 5.14 is translated into an SBB with the name “A1”. From the BPEL process in Figure 5.14, the main SBB and four further SBBs, one for each flow branch, are generated. During the processing of the Java code representing the main sequence of the BPEL process, the “flowA” activity is reached first, and request events are sent to activate the SBB A1 and SBB A2. Both SBBs now process their tasks simultaneously. The main SBB is activated again, when both SBBs reply with a response event. If both responses are received, the program code for the second flow, “flowB”, can be reached. Here, the SBB B1 and SBB B2 are called with request events, and they answer with response events. Afterwards, the Java code that represents the remaining activities of the main sequence can be processed.

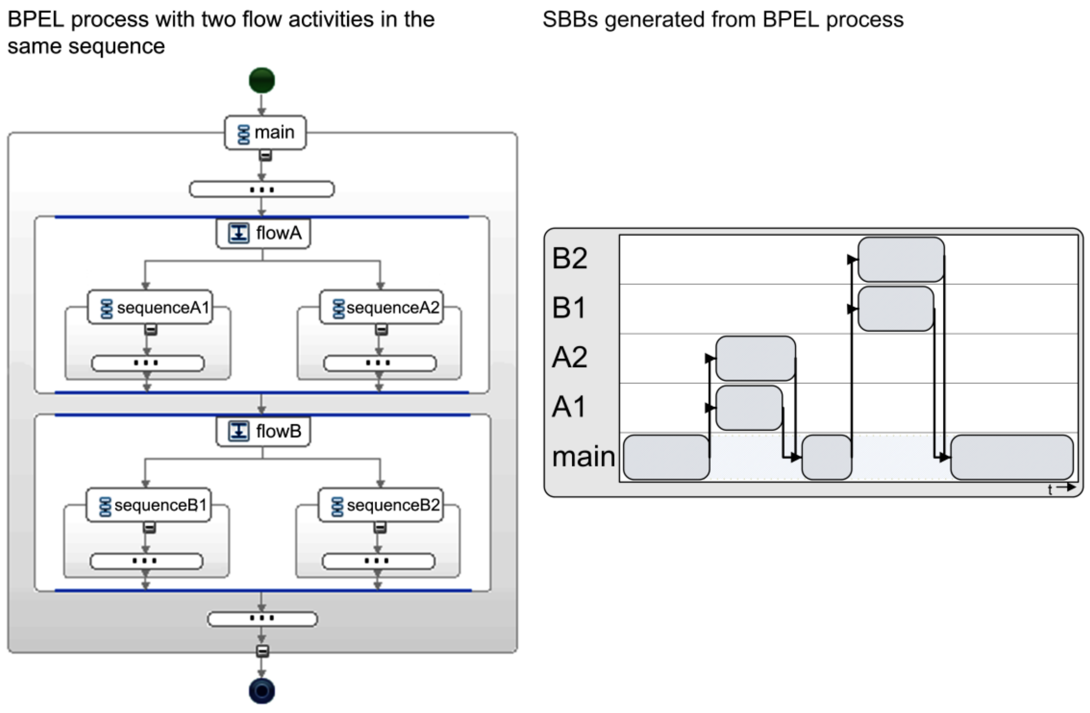

BPEL process with two flow activities in the same sequence
SBBs generated from BPEL process
main
flowA
sequenceA1
sequenceA2
flowB
sequenceB1
sequenceB2
B2
B1
A2
A1
main
t

Figure 5.14: BPEL process with two flow activities within a sequence and the resulting SBBs

As already mentioned above, Flow activities can be nested within Flow activities in BPEL. Figure 5.15 shows such a BPEL process.

“FlowB” is embedded in the right flow branch of “flowA”. A SBB is generated for both flow branches of “flowA”. In this example, the SBBs are named as “A1” and “A2” (on the right side of Figure 5.15). In contrast to the flow activities which are contained within the same sequence, the request events are sent by the SBB, in which the called flow is contained. In the example shown in Figure 5.15, the SBB A2 calls the SBB B1 and SBB B2, so the SBB A2 sends the request events and expects the response events. Therefore, SBB A2 can finish its processing only when SBB B1 and SBB B2 have sent their response events. The main SBB can continue with the processing when SBB A1 and SBB A2 are finished, and the response events are received by the main SBB.

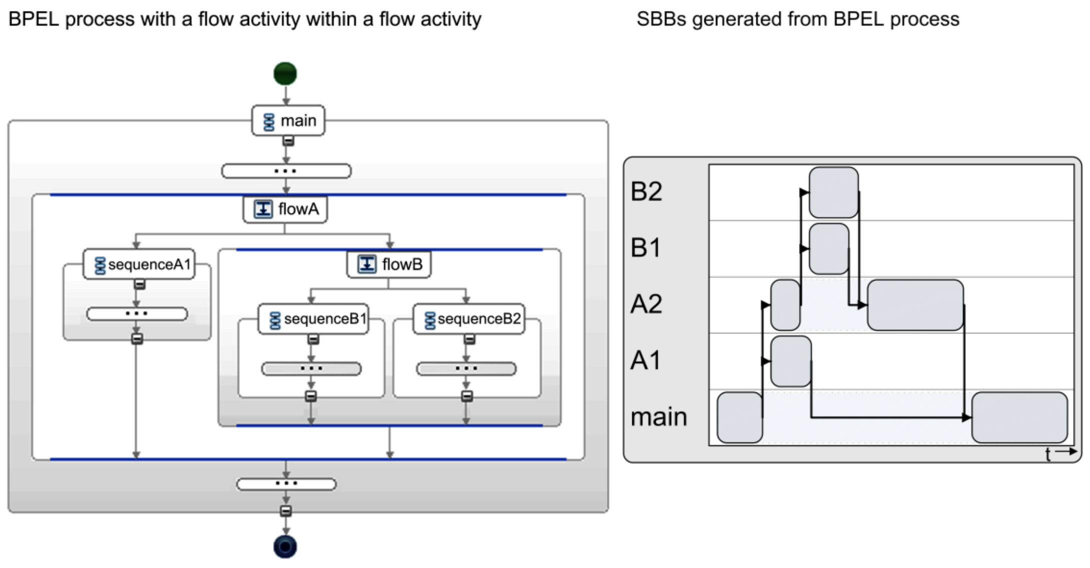

BPEL process with a flow activity within a flow activity
SBBs generated from BPEL process
main
flowA
sequenceA1
flowB
sequenceB1
sequenceB2
B2
B1
A2
A1
main
t

Figure 5.15: BPEL process with a flow activity nested within another flow activity and the resulting SBBs

This approach has also been used in the TeamCom project. The prototypical implementation of the "Single SBB concept" together with the Code Generator (refer to section 5.2.1) approach was modified to support the new "Parallel Program Flow concept". The "Parallel Program Flow concept" is a modification of the "Single SBB concept". New SBBs are generated for the parallel parts of a BPEL process. The Java classes are built from a number of code snippets. The code generator is required to generate and compile the SBBs, and build the service and the deployable unit.

The support of new protocols and functionalities is hindered by the use of these code snippets. It has turned out that the modifying of code snippets is extremely complex and not applicable. Only the Code Generator concept (refer to section 5.3.1) is applicable with this approach.

In the next two sections, the idea of the flow SBBs will be expanded. A SBB will be defined for each activity of the BPEL process. For instance, every existing activity in BPEL will be represented by a generated SBB that represents this BPEL activity in

JAIN SLEE. Such SBBs are called “activity SBBs”. Two composition approaches are investigated. The first “Orchestration concept” (refer to section 5.4.3), requires a central control SBB, which contains the common state machine and all parameters. The control SBB controls all activity SBBs and the communication among the SBBs is carried out via the control SBB. In the second “Choreography concept”, no special control SBB is needed and the control is distributed among the SBBs (refer to section 5.4.4). These self-controlled SBBs communicate directly with each other.

5.4.3 Orchestration Concept

The “Orchestration concept” (Eichelmann *et al.,* 2010) is based on the orchestration definition (refer to section 2.1) of web services in Service-Oriented Architectures (SOA).

The “Orchestration concept” is extended in respect to the SOA and not applied on services, but on the components (SBBs) of a service. The workflow of the BPEL process is subdivided into its activities. Each activity of the process is implemented into one SBB. A special control SBB is used to control the service workflow and to coordinate the SBBs of the service. The control SBB instantiates the required SBBs, sets the required parameters, and defines the events on which the SBB can listen and the events which are fired from the SBB. The control SBB contains a state machine to decide which SBB should be called next. The state machine is derived from the BPEL process and generated by the service description parser of the Runtime Service Composition approach.

In Figure 5.16, the service consists of four SBBs, one control SBB and three activity SBBs (A, B, and C) that implement the workflow of the service.

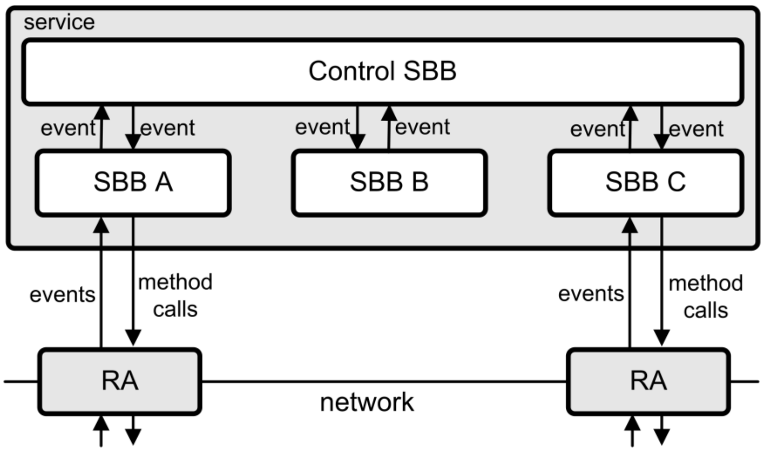


Figure 5.16: Orchestration of SBBs

The control SBB contains a state machine to decide which SBB must be called next. In this case, SBB A is the first SBB, so the control SBB fires an event to SBB A. This event contains the required parameters and information about the RAs used (refer to section 5.2.3). The SBB A executes its tasks, e.g., communicating with a RA, by calling methods from the RA interface. After finishing these tasks, an event is returned back to the control SBB to signal the ending of the tasks. When the control SBB receives the event from SBB A, the control SBB generates an event for the next SBB, according to the state machine. This procedure is repeated, until the workflow is completed.

For each step within the service workflow, the control SBB is required and called. The “Orchestration concept” is suited in all cases where a central point of control is required. For example, in the services defined for the proposed framework (refer to section 7.2) this approach will be required for the control of the service creation, configuration, removing, and monitoring. All of these tasks require central control structures.

The drawback of this approach is that the control SBB is involved in every workflow step, which is usually not required in the normal service execution. In the “Choreography concept” presented in the next section, the SBBs are able to interact directly with each other, which will reduce overhead in comparison to the “Orchestration concept”.

5.4.4 Choreography Concept

The “Choreography concept” (Eichelmann *et al.,* 2010) is based on the definition of choreography of web services (refer to section 2.1) in Service-Oriented Architectures (SOA).

Also the “Choreography concept” is extended and not applied on services but on the components (SBBs) of a service.

In the “Choreography concept”, the BPEL workflow is subdivided into multiple parts. For each BPEL activity, one SBB is defined. For the service execution, no central state machine is required. The SBBs of the “Choreography concept” are not orchestrated by a control SBB. Each SBB fulfils the tasks which are described in the corresponding BPEL activity. Each SBBs knows its own tasks and its communication partners.

A SBB starts to operate after it has received an event from its respective predecessor. This event includes all required parameters. Thus, the SBB can start to process its task after having received the required event. The tasks the SBB has to perform are implemented within the SBB and can be configured at the start time of the service. After the execution of the SBB tasks have been finished, an event will be prepared and fired to the successor SBB(s) of the service.

Figure 5.17 illustrates an example service using the “Choreography concept”. This service consists of three SBBs (A, B, and C) and two resource adaptors. The three SBBs are generated by the Code Generator or by the service description parser of the Runtime Service Composition approach.

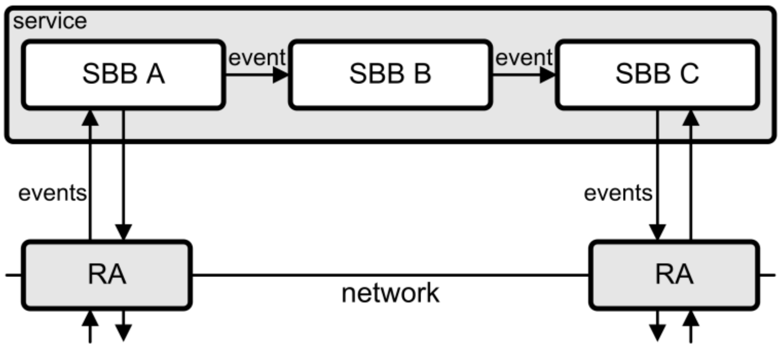


Figure 5.17: Choreography of SBBs

The service is activated when the SBB A receives an event from a resource adaptor. SBB A communicates with the resource adaptor, executes its part of the workflow, and fires a new event to the next SBB, in this case, to SBB B. After SBB B has executed its part of the workflow (e.g., copy and set parameters), this SBB will fire an event to the next SBB. The tasks of SBB C include the communication with a resource adaptor.

In contrast to the “Orchestration concept”, the “Choreography concept” requires no central control SBB. Each SBB handle its own part of the BPEL workflow. Both of the approaches for service creation presented, the Code Generator (refer to section

5.3.1) approach and the Runtime Service Composition (refer to section 5.3.2) approach, can be applied in combination with the “Choreography concept”. In the proposed framework, the Service Execution Environment (SEE) uses the “Choreography concept” for service execution (refer to section 7.2.3). Once the service execution has been triggered, the SBBs of the service will be able to communicate directly with each other. Each SBB knows its communication partners. Therefore, they can communicate directly without the communication overhead which is produced by involving a central SBB.

5.5 Conclusion

This chapter presented several approaches for service description, service creation, and service execution. For these approaches, different concepts were proposed and analysed.

The first section discussed the possibility of how to describe a service. As the result of chapter 3, BPEL (refer to section 3.6) fulfils the required criteria and offers the most possibilities for the description of value-added services. Therefore, this thesis uses BPEL as description language. It was shown how BPEL can be used to describe services. On the one hand, it was explained how the service logic of the service can be described in BPEL and which BPEL language elements are available for this description. On the other hand, it was discussed how the BPEL partner links can be used to describe the functionalities and resources that are required for a service.

The second section introduced two concepts for this service creation process, the Code Generator concept and the Runtime Service Composition concept. In case of the Code Generator concept, the service is generated from code snippets, compiled and packed into a deployable unit. In case of the Runtime Service Composition concept, the service is generated from SBBs at start time of the service. These components represent the service logic and the service functionality. The second concept allows an easy possibility to add new functionality to the framework and offers easy possibilities for third-party development. It also offers some advanced features for monitoring and modifying the service at runtime. Because of its advantages and its flexibility, the Runtime Service Composition approach was selected for this work.

In the third section, four service execution concepts were analysed. The first concept, the “Single SBB concept”, tries to implement all of the service logic in one SBB (refer to section 5.4.1). For this approach, a prototype of the Code Generator was implemented, which was also used in the TeamCom research project (TeamCom, 2010). This approach shows the constraint that it does not support parallel program execution.

When applying the second concept, the “Parallel Program Flow concept”, (Eichelmann *et al.,* 2009) separate SBBs will be generated for all parts of the service that require parallel program execution (refer to section 5.4.2). The prototype of the Code Generator was modified to support multiple SBBs. However, both concepts show problems with the extensibility and the integration of new functionalities. It is very difficult for third-party developers to add the support of new protocols to the framework.

The other two concepts, the “Orchestration concept” (refer to section 5.4.3) and the “Choreography concept” (refer to section 5.4.4), offer, in contrast to the previous approaches, a good extensibility. For the proposed framework, the “Orchestration concept” has been selected for service creation, configuration, removal, and control (refer to section 7.2.2). Its main advantage is the common control SBB that allows a centralised control point for service composition and management. However, a service structure with a central control SBB has disadvantages for the execution of services, since it would be involved in all communications among any SBBs. This would cause a high processing overhead. Therefore, the “Choreography concept” was selected for the service execution (refer to section 7.2.3). The advantage here is the decentralised control of the service components.

6 Proposed Framework

This chapter presents the architecture of the framework. It describes the various elements in more detail and begins with an overview of the architecture and its elements (refer to section 6.1). The architecture is divided into two main parts, the Service Creation Environment (SCE) and the Service Execution Environment (SEE).

Section 6.2 takes a closer look at the SCE. The SCE provides the developer with tools to describe the services. It offers Communication Building Blocks (CBBs), a graphical service management tool, a marketplace, a repository, and a graphical development tool.

The SEE is the runtime environment for services it is described in section 6.3. The SEE controls the services and the service's instances and consists of a layered structure with three layers, the service management layer (refer to section 6.3.1), the service execution layer (refer to section 6.3.2), and the resource connection layer (refer to section 6.3.3).

6.1 Architecture Overview

To fulfil the requirements defined in section 5.1, a framework is defined that offers an automated solution for the creation, provisioning, and execution of value-added telecommunication services. The architecture supports formal BPEL (refer to section 5.2) service descriptions widely used in the IT sector. For the service execution environment the JAIN SLEE technology (refer to section 4.4) is selected. It is a well-established framework in the telecommunication sector and offers the required performance (refer to section 4.4 and section 4.9). With its Resource Adaptor concept, the support of new protocols is possible. The high complexity of JAIN SLEE is hidden to the service developer, since the service creation is done with BPEL. Therefore, JAIN SLEE has been selected as service execution environment and BPEL as service creation environment.

An overview of the proposed framework is presented in Figure 6.1. The framework consists of the SCE and the SEE. The SCE offers the possibility to describe the service with a service description tool (refer to section 6.2.2). The developer can choose between an existing BPEL development tool, an XML file editor, and a web interface for describing the services. To define the functionalities that are required for the service, e.g., the communication with other services, or mathematical calculations (refer to section 5.2.3), the novel concept of Communication Building Blocks (CBBs) were defined (refer to section 6.2.1). These CBBs offer a mapping between the formal BPEL description of the service functionalities as BPEL partner links and the implementation of these functionalities in the SEE (refer to section 6.3.3).

With the introduction of a new marketplace and the repository (refer to section 6.2.4), it is possible to download service descriptions, CBBs, and Resource Adaptors (RAs) from the Internet and store them in the repository. The user of the framework can create value-added services using the downloaded service descriptions and CBBs. In this case, no knowledge about BPEL, JAIN SLEE, or the underlying protocols is required. Furthermore, complete services may be downloaded, modified and used as base for customised own services.

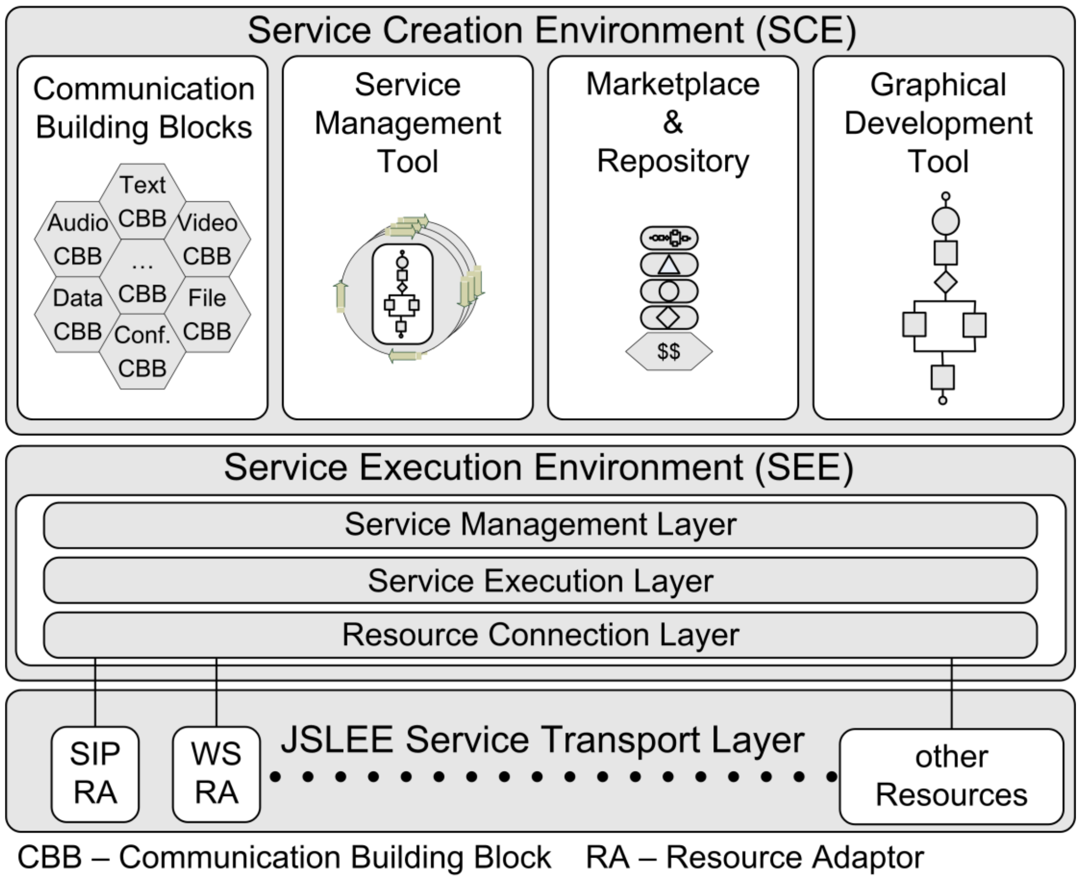

Service Creation Environment (SCE)
Communication Building Blocks
Text CBB
Audio CBB
Video CBB
… CBB
Data CBB
File CBB
Conf. CBB
Service Management Tool
Marketplace & Repository
$$
Graphical Development Tool
Service Execution Environment (SEE)
Service Management Layer
Service Execution Layer
Resource Connection Layer
JSLEE Service Transport Layer
SIP RA
WS RA
other Resources
CBB – Communication Building Block
RA – Resource Adaptor

Figure 6.1: Framework architecture

The SEE is based on JAIN SLEE. For this research project, the JAIN SLEE framework has been extended. The extended framework includes the concepts of JAIN SLEE but add further ideas to the framework. This research project adds a layered structure with three layers (refer to section 6.3) on top of the JAIN SLEE SBB container (refer to section 4.4). Each defined layer fulfils specific tasks.

The first layer is responsible for the management of the framework and of the services (refer to section 6.3.1). It is controlled by the service management tool (e.g., a web interface) and it controls the life cycle of the value-added services and offers the functionality to start and stop the services. This layer is called "Service Management Layer".

The second layer is responsible for the processing of the service logic (refer to section 6.3.2). It realises and executes the logic that was described in the service description. This layer is called the "Service Execution Layer".

The third layer is called the "Resource Connection Layer" (refer to section 6.3.3). It implements and executes the functionalities of the CBBs. These functionalities are described in the service description as partner links.

To communicate with other applications, or to receive and send events between JAIN SLEE and other applications and resources, two possibilities are defined in the JAIN SLEE specification, namely the RA concept and the EJB concept (refer to section 4.4). Both concepts are supported in this framework (refer to section 7.1.3) and are handled by the resource connection layer (refer to section 6.3.3).

The SEE utilises the JAIN SLEE Resource Adaptor concept to offer the support for new protocols. The RAs are part of the JAIN SLEE Service Transport Layer.

For each of the layers on top of the JAIN SLEE SBB container, special components are defined (refer to section 7.1.5). These components are based on the JAIN SLEE SBBs. Figure 6.2 offers an overview of the different SBBs and their corresponding layers. The SBBs defined in the management layer are called "Management SBBs" (MSBBs). They are responsible for the control of the framework and the services (refer to section 6.3.1). The SBBs in the Service Execution Layer are called "Logic SBBs" (LSBBs); these SBBs are responsible for the service logic and are mapped from the BPEL activities (refer to section 6.3.2). The SBBs in the Resource

Connection Layer are called “Resource Connection SBBs” (RCSBBs). These SBBs are part of the CBBs and implement the methods containing the functionality that can be used within the services (refer to section 6.3.3).

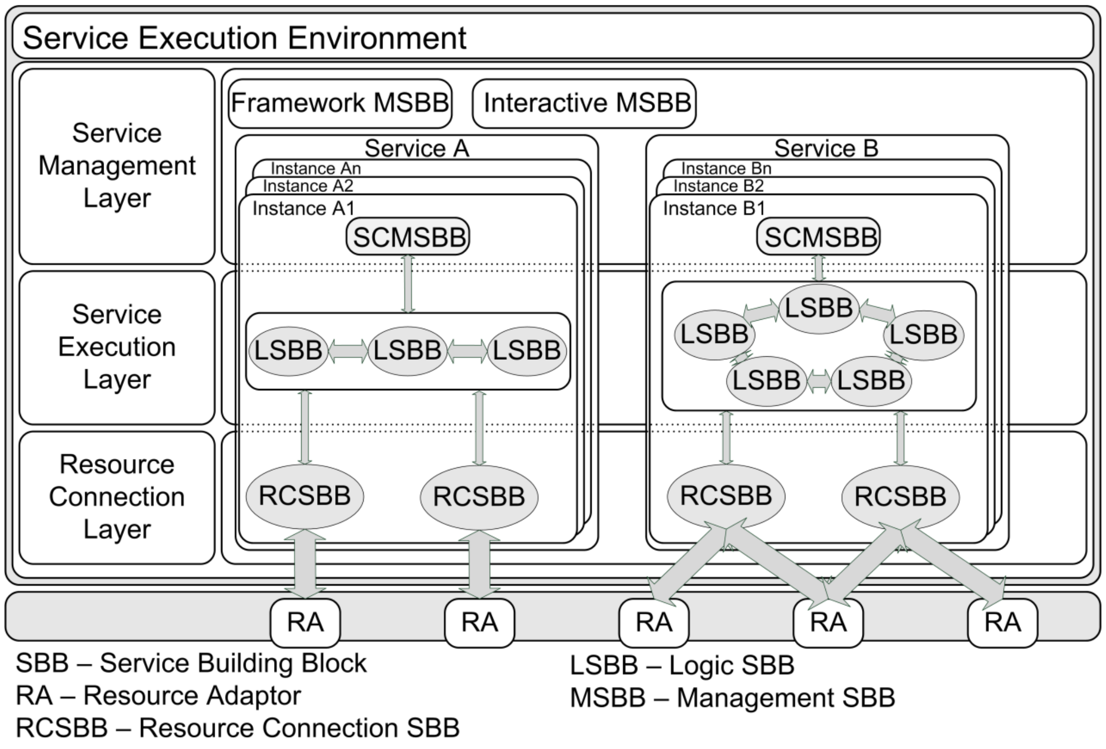


Figure 6.2: Service execution environment

The SEE supports multiple instances of a service. One service instance can contain components from all layers. The SEE supports the creation, the execution, the removal, and the reconfiguration of service instances. These responsibilities are handled by the service life cycle phases (refer to section 7.2).

The communication between the components is provided by a novel communication channel concept. This concept is part of the extensions, of JAIN SLEE. These communication channels are based on the JAIN SLEE event concept but offer a more flexible handling of the communication and define special events for the communication of the framework components (refer to section 7.1.4).

The extended SEE offers a central point to store the management and monitoring information, references to the service instances, components, and variables. This central point is denoted as “framework context” (refer to section 7.1.1).

For the variable types (refer to section 7.1.2) required in the services, a mapping is defined which is also part of the CBBs (refer to section 6.2.1). It is possible to add new variable types to the framework by adding a new appropriate CBB.

6.2 Service Creation Environment

The main task of the SCE is to describe the value-added services. The SCE offers tools that are required to describe the services and, furthermore, web interfaces which allow managing the framework and the services (Figure 6.3). A BPEL development tool (refer to section 6.2.2) is used for the description of the service.

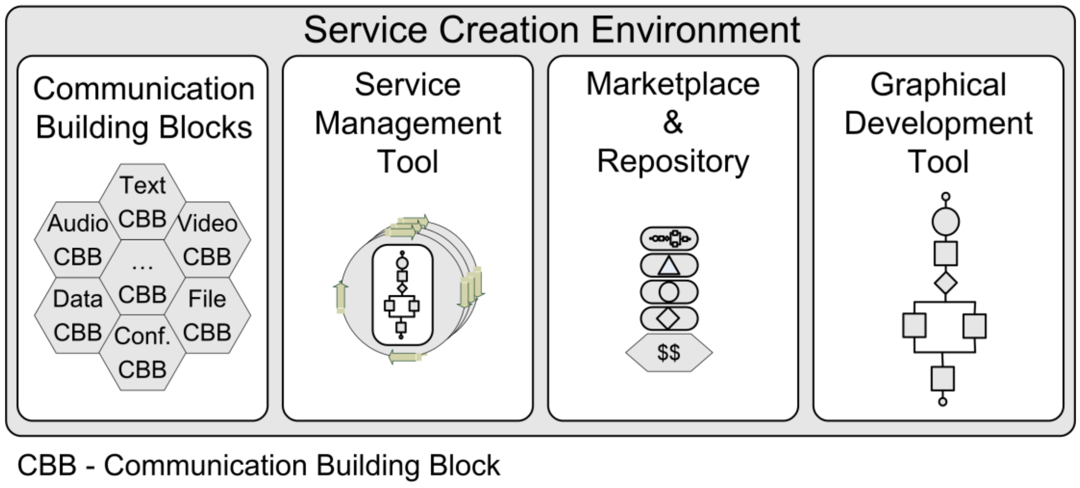

Service Creation Environment
Communication Building Blocks
Text CBB
Audio CBB
Video CBB
… CBB
Data CBB
File CBB
Conf. CBB
Service Management Tool
Marketplace & Repository
$$
Graphical Development Tool
CBB - Communication Building Block

Figure 6.3: Service creation environment

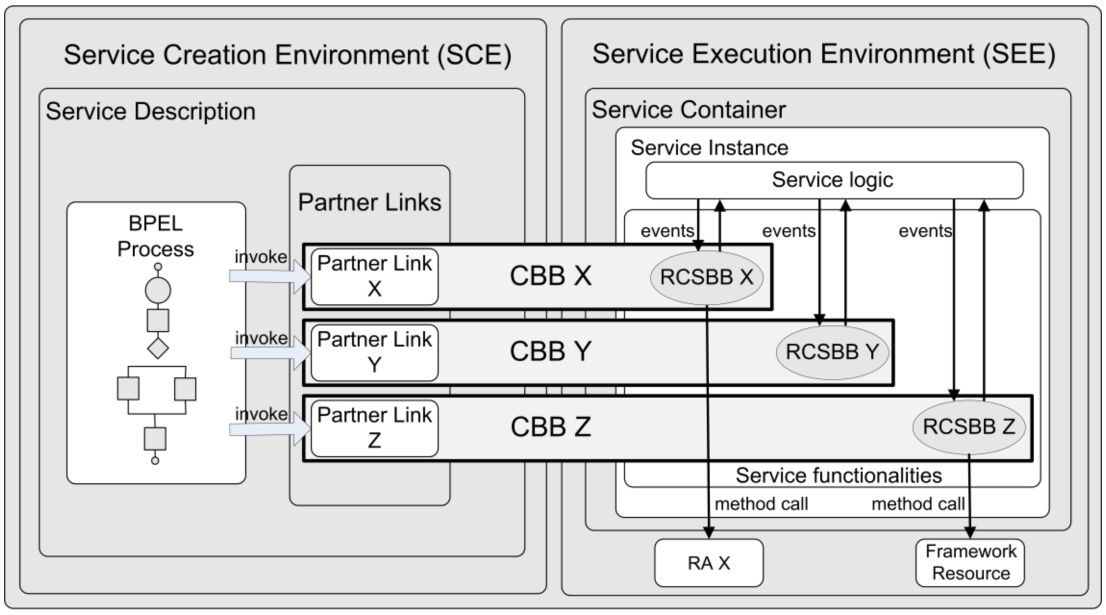
The service designer develops the service logic with the desired tools and chooses the required functionalities, which are pre-defined in the form of Communication Building Blocks (CBB) (refer to section 6.2.1). The complete service description is then handed over to the SEE. This step is supported by the Service Management Tool (refer to section 6.2.2). This tool provides web interfaces for managing and monitoring the services. Service instances can be generated, started, and stopped, and the status of service instances can be controlled and monitored.

Furthermore a marketplace (refer to section 6.2.4) and a service repository are provided. The marketplace is a web application that offers the possibility to search for new service descriptions, new CBBs, and resource adaptors. The marketplace web interface connects to a marketplace server, which can be operated by the framework provider or some other community. The repository allows the developer to store the developed or downloaded service descriptions.

6.2.1 Communication Building Blocks

The proposed framework introduces the novel concept of Communication Building Blocks (CBBs). The CBBs define the mapping from the description of functionalities to their implementation (Figure 6.4). They offer the implementation of the functionalities available in the SEE and the description of these functionalities in the SCE.

The service developer can describe the functionalities with the help of the BPEL partner links. A CBB defines the mapping of the functionalities offered in the SEE and the description of these functionalities in the SCE. Figure 6.4 shows the representation of the CBBs in BPEL (SCE) on the left side and the representation of the CBBs in the SEE as Resource Connection SBBs (RCSBBs) on the right side.



CBB – Communication Building Block
BPEL – Business Process Execution Language
RCSBB – Resource Connection Service Building Block
RA – Resource Adaptor

Figure 6.4: Communication Building Blocks (CBBs)

In BPEL, the CBBs are represented by partner links, and the methods from the CBBs are invoked from the BPEL process. On the right side, the CBBs are implemented in SBBs of the SEE. These SBBs are called “Resource Connection SBBs” (RCSBBs). RCSBBs can use RAs for handling the protocol communication with external functionalities and resources such as database access and media server control.

The partner links are described using Web Service Description Language (WSDL) (W3C, 2007b). WSDL describes the interfaces of web services and is independent of platform, programming language, and protocol. WSDL is a meta-language that allows the description of the offered functionalities, data types, and data exchange protocols of web services in the form of WSDL files.

For each CBB, a WSDL partner link description exists. This description contains the methods and parameters offered by the CBB. In the BPEL process, the desired method can be selected using the partner links. Since, in the proposed framework the services are not realised as web services (refer to section 5.2), web service implementation of the WSDL descriptions are actually not needed.

The taken approach maps the operations from the partner links to Java methods offered by the RCSBBs (refer to section 6.3.3). These RCSBBs implement the methods that are described in the partner links. For each partner link in BPEL, a RCSBB exists, which implements the functionality described in this partner link. A CBB consists of the BPEL partner links, the corresponding RCSBB, and the required variable types.

A service that wants to use a special functionality from a CBB has to contact the corresponding RCSBB that offers this functionality. The service calls the desired method, assigns the required parameters, and sends an event with this information via the communication channel (refer to section 7.1.4) to the RCSBB.

Self-developed or new functionalities and resource adaptors can easily be integrated and used within the framework by defining a new CBB. The developers of a CBB have to implement the RCSBB part of the CBB with the interfaces and parameters that are required for the communication with the services and the resources. Furthermore, the variable types for non-standard, complex data types (refer to section 7.1.2) have to be implemented. The service description part of the CBB has to be described as partner link. The CBBs can vary in the level of granularity, and coarse-grained CBBs may be preferred to avoid unnecessary complexity.

The advantage is that new classes and WSDL partner links have to be implemented only once. Then, the functionalities of these classes can easily be used in the BPEL process. The BPEL developer does not need to take care about the underlying protocols, because the implemented CBBs already considered that. The BPEL developer simply selects the desired functionalities from the pre-defined partner links.

Depending on the external resources, the communication between the CBBs and the resources can be very complex, but this complexity is also hidden to the service developers.

To generate a service that uses functionality from a CBB, the corresponding RCSBB has to be deployed on the application server. For each CBB used within the service, an instance of the corresponding RCSBB is created. The RCSBB itself has to be deployed on the AS before the service using the CBBs, is called.

6.2.2 Graphical Development Tools

The graphical development tools are part of the Service Creation Environment. All services use BPEL as description language (refer to section 5.2). Services can be designed from scratch, an existing service description can be modified, services from the repository or acquired from the marketplace (refer to section 6.2.4) can be composed and integrated.

The developer can choose between a graphical development tool, an XML editor, and a web-based development tool (Figure 6.5).

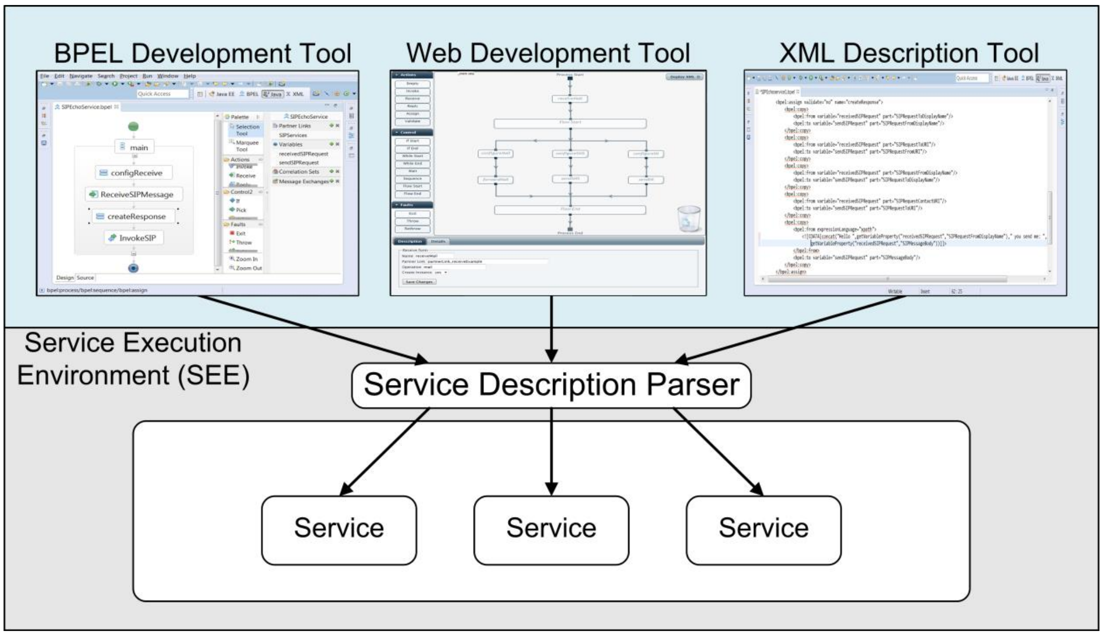


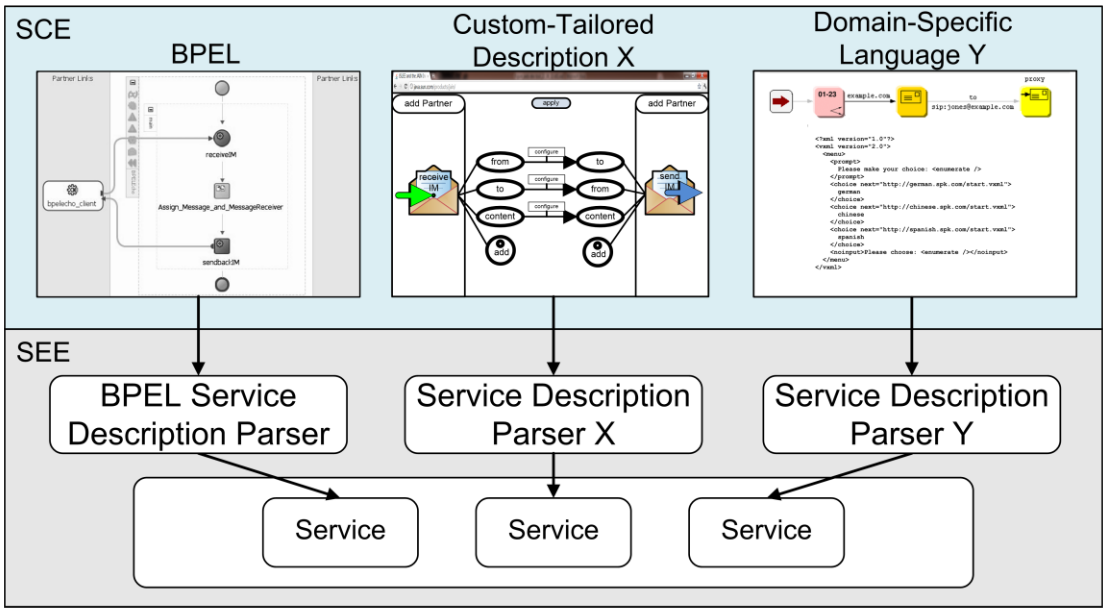
Figure 6.5: BPEL-based development tools

Available BPEL development tools generally offer both a graphical user interface and the possibility to edit the XML BPEL process document directly with an XML editor, e.g., the Eclipse BPEL Designer (Eclipse 2013). Both possibilities can be used to develop a BPEL process, and the developer can switch between both possibilities.

The web-based tool is a specially designed BPEL development tool. This tool offers specific capabilities to develop BPEL processes and to manage services (refer to section 6.2.3). With this tool, the developed services can be uploaded to the service repository and triggered for execution. The web-based development tool is able to interact with the service management (refer to section 6.3.1). This allows the developer to monitor running service instances and service components. Furthermore, the services that are currently executed can be reconfigured and modified (refer to section 7.2.4).

The developed service description is uploaded to the service description parser which is part of the service management (refer to section 6.3.1). It analyses the service description and collects information for the generation of the service.

Here BPEL is used for service description, but generally, it would be possible to use other service description languages. Figure 6.6 illustrates an example that uses other service descriptions. In this case, suitable service description parsers have to be added to the SEE.



```
<?xml version="1.0"?>
<vxml version="2.0">
  <menu>
    <prompt>
      Please make your choice: <enumerate />
    </prompt>
    <choice next="http://german.spk.com/start.vxml">
      german
    </choice>
    <choice next="http://chinese.spk.com/start.vxml">
      chinese
    </choice>
    <choice next="http://spanish.spk.com/start.vxml">
      spanish
    </choice>
    <noinput>Please choose: <enumerate /></noinput>
  </menu>
</vxml>
```

Figure 6.6: Different service descriptions and parsers

For each supported description language a suitable service description parser is required. This allows the framework provider to develop service description tools that are tailored to the customers' needs and support a wide range of possible service descriptions.

6.2.3 Service Management Tool

For the control of the service management, the service management tool is required. This tool is a web interface to control the service management within the SEE (refer to section 6.3.1). With this tool, the user has the ability to send instructions, e.g., start, stop, and remove services and service instances. It allows the user to manage and monitor information about the status of the services and the framework. The management tool can be implemented as web application (e.g., as servlet).

The web application offers the possibility to choose the desired service description (Figure 6.7) from the file system or from the repository (refer to section 6.2.4). The selected service description is transferred to the service management of the SEE, where the service is parsed. Service descriptions, which are loaded from the file system, are stored in the service repository.

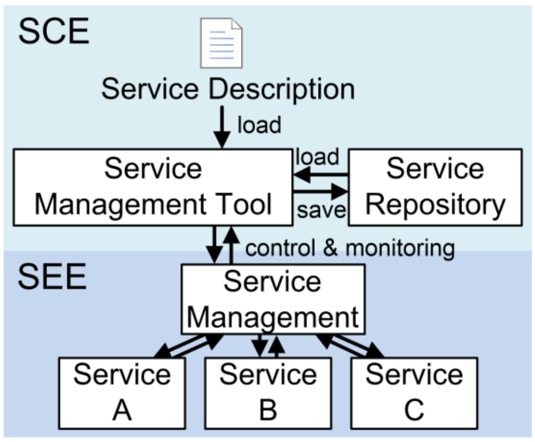


Figure 6.7: Service management tool

With the service management tools, services or service instances can be monitored and controlled. Furthermore, the status of individual service components of a service instance can be monitored.

6.2.4 Marketplace & Repository

The marketplace offers the download of pre-defined service descriptions and components like resource adaptors and CBBs from a marketplace server. The marketplace can be bound to one component provider or be an open marketplace. An open marketplace enables opportunities for third-party developers to offer their service descriptions, RAs, and CBBs. The repository holds all the developed or acquired service descriptions and service components. It is linked with the service management tool (refer to section 6.2.3) that controls the services.

The overview given in Figure 6.8 shows how the marketplace & repository can be used to add new resources to the framework. To support, e.g., new devices, the device developer provides the required CBBs. These CBBs include, on the one hand, the BPEL partner links and, on the other, the RCSBBs with the implemented functionality (refer to section 6.2.1). With these CBBs, the service developer can describe services that use the new functionalities.

To develop and execute the services, the required CBBs have to be available in the framework. The partner links of the CBBs are required for the SCE to describe a service, whereas the RCSBB parts of the CBBs have to be deployed in the SEE to execute a service.

The communication with external resources is performed with RAs (refer to section 6.3.3). New RAs can be obtained from the marketplace, e.g., from a device manufacturer or from a third-party developer.

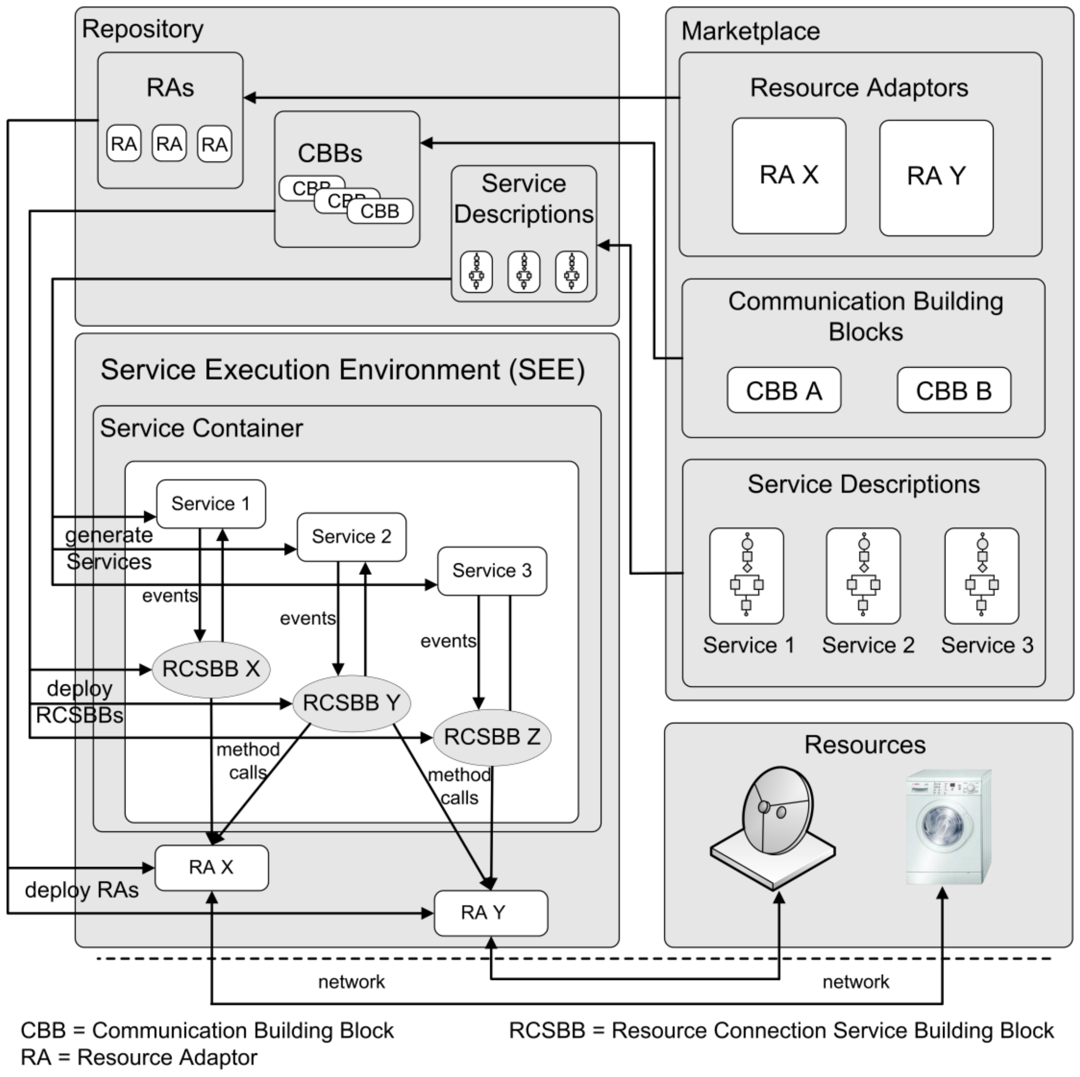


Figure 6.8: Marketplace and repository: integration of external resources

In addition, the marketplace can offer service descriptions. These service descriptions can be stored in the repository. With the service management tool, services can be triggered for creation and execution, or they can be transferred to the service developer tool for modification.

Service descriptions may require CBBs or RAs that are not available within the repository. In this case, the service developer has to acquire the required CBBs and RAs from the marketplace.

A device manufacturer can provide the customer with a complete bundle that includes everything that is needed for using the devices. This bundle may include the required CBBs. The customer can deploy the RAs and RCSBBs and execute the services.

The marketplace offers the great advantage that the support of new resources can easily be added to the framework. For example, to add the support of a new protocol to the framework, the required RA together with the corresponding RCSBB is required. These components can be acquired from the marketplace. Service descriptions that use the new resources can also be downloaded from the marketplace and stored in the repository.

6.3 Service Execution Environment

As result of chapter 4, the JAIN SLEE framework has been selected as basis for the SEE. However, the JAIN SLEE framework needs to be extended to support the service generation at runtime (refer to section 5.3.2). The extension offers capabilities for automated composition and management of the services, for integration of functionalities defined in the CBBs into the services, and for the execution of the generated services.

For these purposes, a layered structure has been proposed on top of JAIN SLEE (Figure 6.9). This layered structure consists of three layers, the Service Management Layer (refer to section 6.3.1), the Service Execution Layer (refer to section 6.3.2), and the Resource Connection Layer.

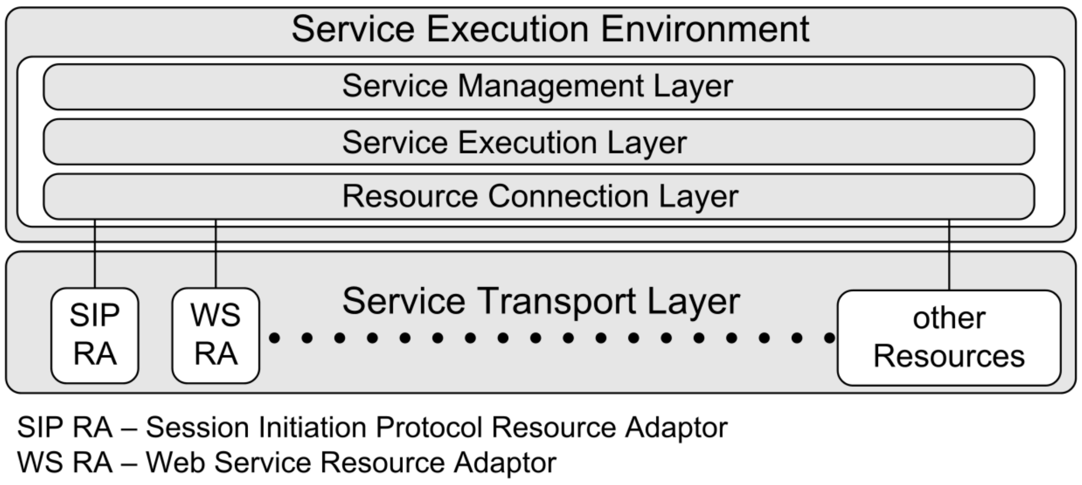

Service Execution Environment
Service Management Layer
Service Execution Layer
Resource Connection Layer
Service Transport Layer
SIP RA
WS RA
other Resources
SIP RA – Session Initiation Protocol Resource Adaptor
WS RA – Web Service Resource Adaptor

Figure 6.9: Layers of the service execution environment

6.3.1 Management Layer

The Management Layer is responsible for the management of the framework and the management of the services. It controls the life cycle of the value-added services and offers the functionality to start and stop the services. The management can also generate new services from a BPEL service description by composing SBBs. The components of the management layer themselves are implemented as SBBs. In order to distinguish them from other SBBs, they are called “Management Service Building Blocks” (MSBBs). The three most important MSBBs are the Framework MSBB, the Service Control MSBB (SCMSBB), and the Interactive MSBB (Figure 6.10).

The responsibility of the Framework MSBB is to control the framework. It creates and manages the framework context (refer to section 7.1.1), which stores status information about the framework and the services, references to the services, service instances, service components (refer to section 7.1.5), and communication channels (refer to section 7.1.4). Furthermore, it manages services and their life cycles; for example, it triggers the creation, begin, end, and reconfiguration, and monitors them.

Another aspect is the communication with the SCE and the Interactive Management SBB. The interface to the SCE is required for receiving service descriptions from the SCE and sending status information to the SCE.

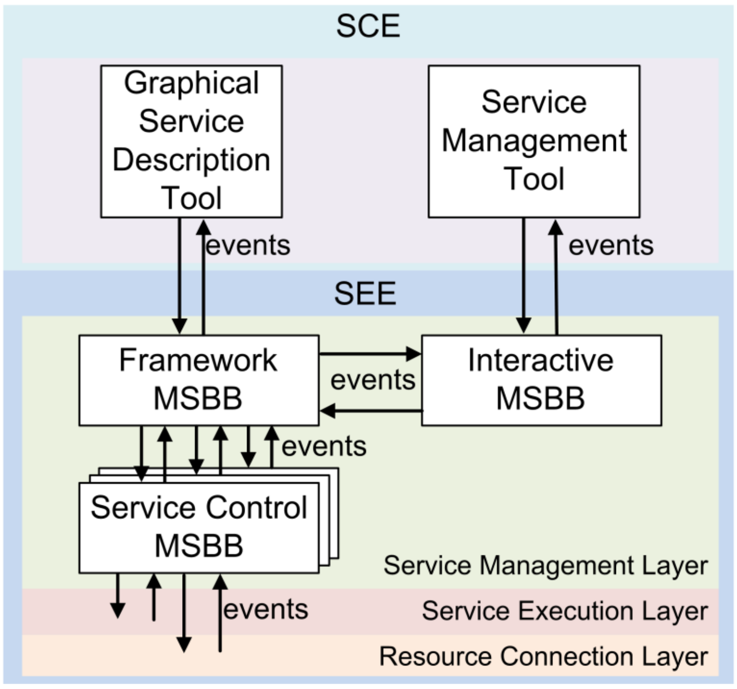


Figure 6.10: MSBBs of the management layer

The Interactive Management SBB offers interfaces for service and framework monitoring information. The Interactive Management SBB waits for user instructions from the Service Management Tool, e.g., start/stop/remove/create service instances, and exchanges this information with the Framework MSBB.

A Service Control MSBB (SCMSBB) of a service instance is required for creation, composition, configuration, monitoring, execution, and life cycle control of the service components (refer to section 7.1.5). One SCMSBB is responsible for one service instance (Figure 6.11).

The SCMSBBs receive their instructions and the service descriptions from the Framework MSBB. Furthermore, a SCMSBB creates and manages the context of a service instance within the framework context (refer to section 7.1.1). The service context stores the information of a service instance, the variable instances, and communication channels (refer to section 7.1.4) in order to be able to communicate with the service components.

Furthermore, each SCMSBB contain a service description parser (refer to section 6.2.2). With this parser, the SCMSBB can analyse the BPEL service description and generate the service components.

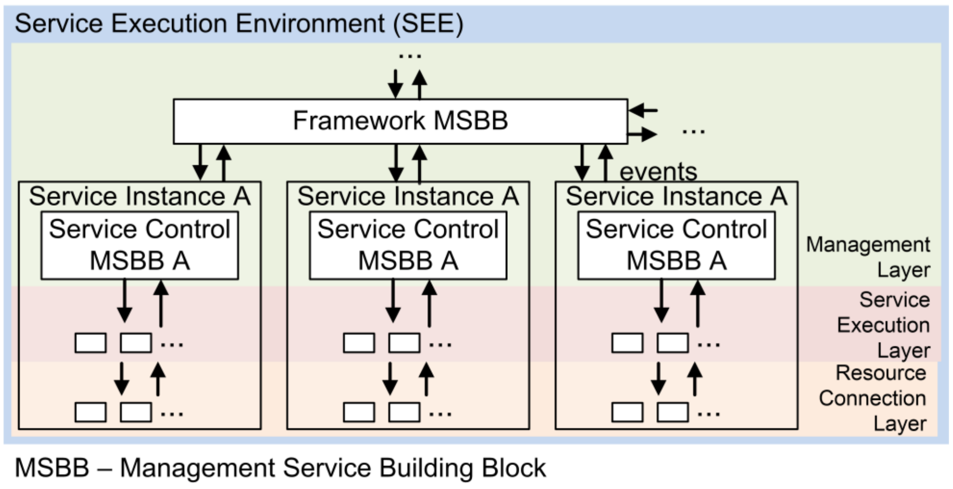


Figure 6.11: Relationship between SCMSBB and service instance

6.3.2 Service Execution Layer

The service logic is located in the Service Execution Layer (Figure 6.12). This logic is represented by the Logic SBBs (LSBBs) (refer to section 7.1.5). The LSBBs realise the service logic defined by the BPEL service description and the workflow of the BPEL process (Eichelmann *et al.*, 2010). They are created, configured, controlled, removed, executed, and monitored by a SCMSBB.

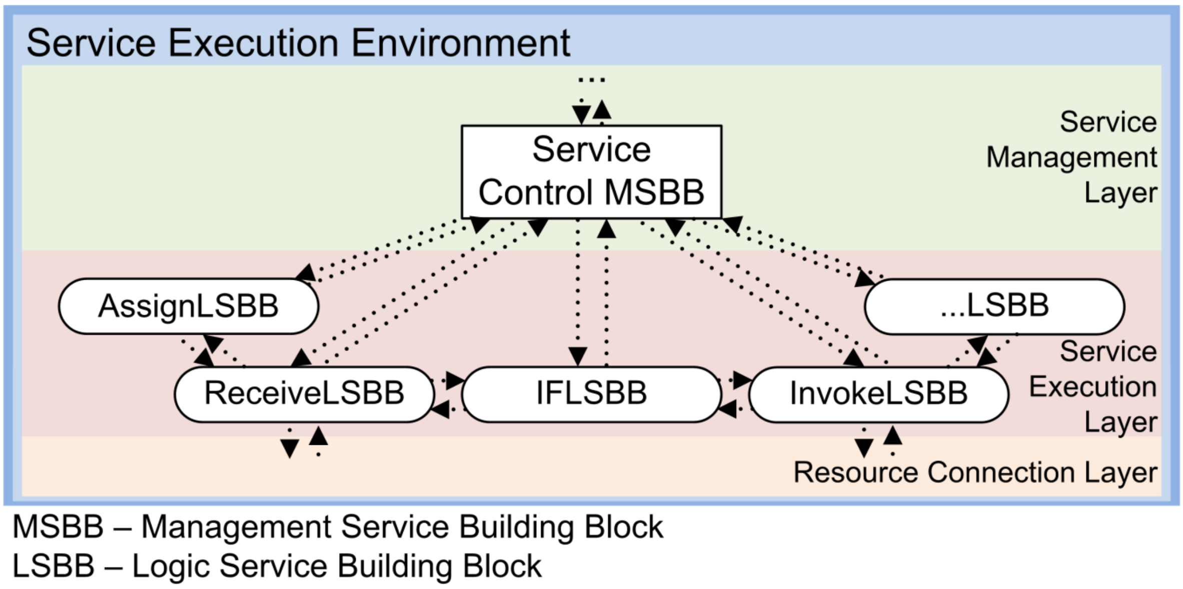

Service Execution Environment
...
Service Control MSBB
Service Management Layer
AssignLSBB
...LSBB
ReceiveLSBB
IFLSBB
InvokeLSBB
Service Execution Layer
Resource Connection Layer
MSBB – Management Service Building Block
LSBB – Logic Service Building Block

Figure 6.12: LSBBs of the service execution layer

The SCMSBB configures the LSBBs by setting the required parameters and the context, and decides on which communication channels (refer to section 7.1.4) a LSBB has to listen for events and on which communication channels it has to fire events. The LSBBs are derived from the BPEL activities (Table 5.1) (refer to section 5.2.2). For each BPEL activity, a corresponding LSBB is defined. In Table 6.1, an overview of all LSBBs is given.

Table 6.1: Overview of the LSBBs and their tasks

LSBB name	**LSBB tasks**
Invoke LSBB	The Invoke LSBB is used to call methods which are implemented in the RCSBBs (refer to section 6.3.3). These methods implement the functionality that is described in the partner links of the BPEL service description. The CBBs map these BPEL partner link calls to the corresponding RCSBB. Which functionality is called on the RCSBB is defined in the BPEL service description. The Invoke LSBB is mapped to the Invoke activity in BPEL.
Receive LSBB	This LSBB listens for events from RCSBBs (refer to section 6.3.3). The Receive LSBB implements the Receive activity from the BPEL service description. The BPEL Receive activity defines the partner link for the communication. The CBBs map this partner link description to the corresponding RCSBB. The Receive LSBB waits for an asynchronous event from this RCSBB.

Reply LSBB	This LSBB allows the service to send an event in reply to an event that was received via a Receive LSBB. The combination of a Receive LSBB and a Reply LSBB forms a request-response operation for the service. The Reply LSBB is mapped to the Reply activity in BPEL.
Assign LSBB	The Assign LSBB can be used to copy data from one variable to another, insert literals into variables, and insert new values into the variables by using expressions. The Assign LSBB is mapped to the Assign activity in BPEL.
Throw LSBB	The Throw LSBB is used when a service instance needs to signal an internal fault explicitly. An event is sent to the SCMSBB of the service instance where the fault is handled. The Throw LSBB is mapped to the Throw activity in BPEL.
Wait LSBB	The Wait LSBB is used to define a deadline or a duration. The Wait LSBB completes if the specified deadline or duration is reached. The Wait LSBB is mapped to the Wait activity in BPEL.
Empty LSBB	The Empty LSBB does nothing: it can be used, e.g., for suppressing a fault that needs to be caught, or for providing a point of synchronization in a flow. The Empty LSBB is mapped to the Empty activity in BPEL.
Extension LSBB	The Extension LSBB is used to define new LSBBs that are not defined in this table. It offers the possibility to add new individual LSBBs to the framework. The Extension LSBB is mapped to the Extension activity in BPEL.
Exit LSBB	To end the service instance immediately, the Exit LSBB is used. The Exit LSBB is mapped to the Exit activity in BPEL.
Rethrow LSBB	The Rethrow LSBB is used to propagate faults. It is applied in fault handlers. The Rethrow LSBB is mapped to the Rethrow activity in BPEL.
Sequence LSBB	A Sequence LSBB contains one or more LSBBs that are executed sequentially in the lexical order in which they appear within the service description of the Sequence LSBB. The Sequence LSBB is finished, when the last LSBB in the sequence is executed. The sequence LSBB is mapped to the Sequence activity in BPEL.
If LSBB	Conditional behaviour is provided by the If LSBB. The If LSBB contains a list of one or more conditional branches defined by the “if” element and the optional “elseif” and “else” elements. The order in the list of branches is also the order in which the conditions are analysed. If a condition is evaluated to true, the corresponding branch is executed; if this condition evaluates to false, the next condition is analysed; if no condition evaluates to true, then the else branch is executed. The If LSBB is completed when the contained LSBB of the selected branch is completed, or is completed immediately when no condition evaluates to true

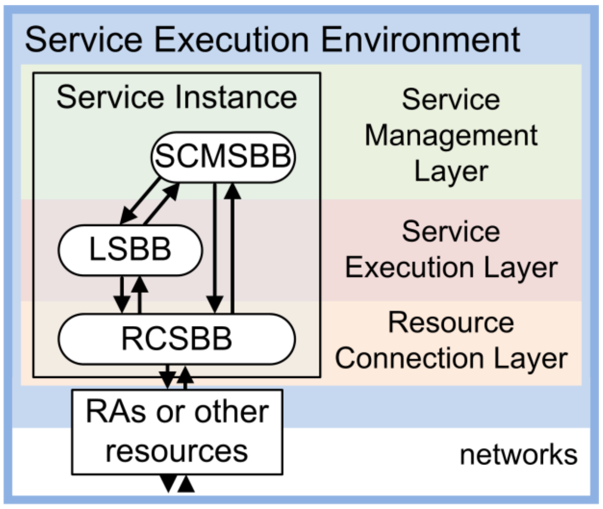
	and no else branch is specified. The If LSBB is mapped to the If activity in BPEL.
While LSBB	The While LSBB offers a mechanism for a repeated execution of the contained LSBB. A Boolean condition is used to check whether the contained LSBB is executed or not. The condition is analysed for all iterations. Only if the condition evaluates to true, the contained LSBB is executed. The While LSBB is mapped to the While activity in BPEL.
RepeatUntil LSBB	The RepeatUntil LSBB offers a mechanism for repeated execution of a contained LSBB. A Boolean condition is used to check whether the contained LSBB is executed or not. The condition is analysed after the iteration. Only if the condition evaluates to true, the contained LSBB is executed again. In contrast to the While LSBB, the "RepeatUntil" loop executes the contained LSBB at least once. The RepeatUntil LSBB is mapped to the RepeatUntil activity in BPEL.
Pick LSBB	The Pick LSBB can receive events from different LSBBs. It waits until one of the events are received, then it executes the LSBB associated with that event. After an event has been received, no other event is accepted by that Pick LSBB. The Pick LSBB is mapped to the Pick activity in BPEL.
Flow LSBB	The Flow LSBB provides parallel execution of LSBBs. It fires events to these LSBBs and waits for events from them. The Flow LSBB is completed after all called LSBBs have been executed. The Flow LSBB is mapped to the Flow activity in BPEL.
ForEach LSBB	The ForEach LSBB represents a loop, which executes associated LSBBs for a specified number of times. This associated LSBB is invoked by events. The ForEach LSBB can execute the associated LSBB in a parallel or sequential order. The ForEach LSBB is mapped to the ForEach activity in BPEL.
Scope LSBB	The Scope LSBB is used to define a nested LSBB context. A scope can have subordinate LSBBs with associated CBBs, variables, and handlers. The Scope LSBB is mapped to the Scope activity in BPEL.
Compensate LSBB	The Compensate LSBB is used to support compensation for inner scopes. It compensates all inner scopes that have already completed successfully. The Compensate LSBB is mapped to the Compensate activity in BPEL.
CompensateScope LSBB	To compensate a Scope LSBB that has already completed successfully, the CompensateScope LSBB is used. The CompensateScope LSBB is mapped to the CompensateScope activity in BPEL.
Validate LSBB	The Validate LSBB is used to validate the values of variables against their associated data definition. The Validate LSBB is mapped to the Validate activity in BPEL.

All BPEL activities have to be mapped on these LSBBs. For example, the Assign activity is mapped to the Assign LSBB.

The SCMSBB from the Service Management Layer composes the LSBBs and RCSBBs to form a service instance (refer to section 7.1.5). One service instance can consist of one or more LSBBs and zero or more RCSBBs. The Service Execution Layer supports multiple service instances for multiple services. Therefore, for each service within the SEE many service instances of this service can be running in parallel at the same time. Each service instance is composed with and controlled by its own SCMSBB. The LSBBs communicate with other LSBBs, with RCSBBs, and with the SCMSBB via the communication channels (refer to section 7.1.4). To integrate the functionalities and resources described in the CBBs into a service, the LSBBs communicate with the RCSBBs from the Resource Connection Layer. An overview of all components of a service is given in section 7.1.5.

6.3.3 Resource Connection Layer

The Resource Connection Layer implements the methods which can be called by the services. Special SBBs called “Resource Connection SBBs” (RCSBBs) implement these methods (Figure 6.13). The methods represent the service functionalities. Typical functionalities are, e.g., video conferencing, chat, voice recognition, and text to speech. Service functionalities can be implemented directly into a RCSBB, or it can be offered by a RA. The RCSBBs are controlled by the SCMSBB.



SBB – Service Building Block
SCMSBB – Service Control Management SBB
LSBB – Logic SBB
RA – Resource Adaptor

Figure 6.13: RCSBB in the resource connection layer

RCSBBs implement the methods that are described by the BPEL partner links. CBBs (refer to section 6.2.1) map the methods from the BPEL partner links to the implementation of these methods in the RCSBBs. The partner links offer the BPEL representation of the available functionalities. When a service requires a special functionality, for example, calling a participant of a conference, the corresponding method, which is described in the partner link handling conferencing issues, has to be selected in the service description (Eichelmann *et al.,* 2008). In the service instance, which is generated from the service description, the LSBBs call the relevant RCSBBs to invoke the implementation of the requested functionalities.

In case that an external resource sends information to the service (e.g., an incoming call), the corresponding RA receives the appertaining protocol message (e.g., SIP INVITE), generates an event and sends this event to the corresponding RCSBB. The RCSBB executes its implemented functionality and generates an event for the corresponding LSBB.

6.4 Conclusion

The proposed framework offers a consistent automated solution for the creation and provisioning of value-added telecommunication services. The presented architecture has introduced the elements of the framework in detail. For the service description, BPEL is used. BPEL is a technology that is well established in the IT sector. To benefit from the advantages of the technologies established in the telecommunication sector, the service is generated and executed in a service execution environment that is based on JAIN SLEE.

In contrast to the conventional service development with JSLEE, the service creation environment offers a simple development of value-added services (refer to section 8.5.2). JSLEE services are developed with Java; the services that are created with the service creation environment are described, i.e. with a graphical BPEL development tool, a XML editor, or a web-based service description tool. In BPEL, the required resources and functionalities are described as partner links. The service developer does not need special knowledge about the underlying protocols. The required functionalities only have to be invoked on the BPEL partner link. Apart from that, describing the services with BPEL using the CBBs is much faster than programming a conventional JSLEE service in Java. For example, developing a simple Chat service in Java requires 3 days; defining the same service with the service creation environment, however, requires only 5 hours (refer to section 8.5).

It is also possible to acquire services and other components like RAs and CBBs from a marketplace of the SCE. This allows third-party developers to offer own resources and services (Eichelmann *et al.,* 2011). New protocols can be supported by providing the corresponding RAs and CBBs. Self-developed functionalities and RAs can be integrated into the framework by defining new CBBs. These CBBs have to define the mapping between the resources in the RCSBBs and the functionalities described in the partner links. Furthermore, the SCE controls the management, reconfiguration, and monitoring of the services and service instances with its service management tool.

The service execution environment is structured in three functional layers (refer to section 6.3). These layers offer the monitoring and control of services and framework and the composition and execution of the services. Services can automatically be composed from the BPEL description with the Runtime Service Composition approach (refer to section 5.3.2). The BPEL workflow describing the service logic is mapped to the LSBBs in the service execution layer. The RCSBBs of the resource connection layer offer the implementation of the functionalities. The service management layer offers the management, composition, configuration, life cycle control, and monitoring of the service instances.

7 Services in the SEE

The previous chapter has introduced the proposed framework with its main elements, the SCE and the SEE. This chapter takes a closer look at the services that are generated and executed within the SEE of the described framework. In the first section (refer to section 7.1), the general structure of a service instance is discussed in detail. The principles of communication, the service components, and the framework context are defined. In the second section (refer to section 7.2), the life cycle of a service instance is analysed, and execution, reconfiguration, and removal of service instances are described.

7.1 Service Structure

A service consists of multiple elements: the framework context (refer to section 7.1.1), types and variables (refer to section 7.1.2), multiple RCSBBs, LSBBs, and the SCMSBB (refer to section 7.1.5). This section takes a closer look at these elements and describes the communication principles between the service components (refer to section 7.1.4) and between service components and external resources (refer to section 7.1.3).

7.1.1 Framework Context

The framework context offers a possibility for the framework and for the services to store their information in one central place. This concept allows to manage the consistence and persistence of the service instances and gives an opportunity for the framework management to directly monitor the status of each service element.

Figure 7.1 illustrates the hierarchical organisation of the framework context. Each service stores its service context into the framework context. Each service instance stores its context into the service context and each service component stores its context into the service instance.

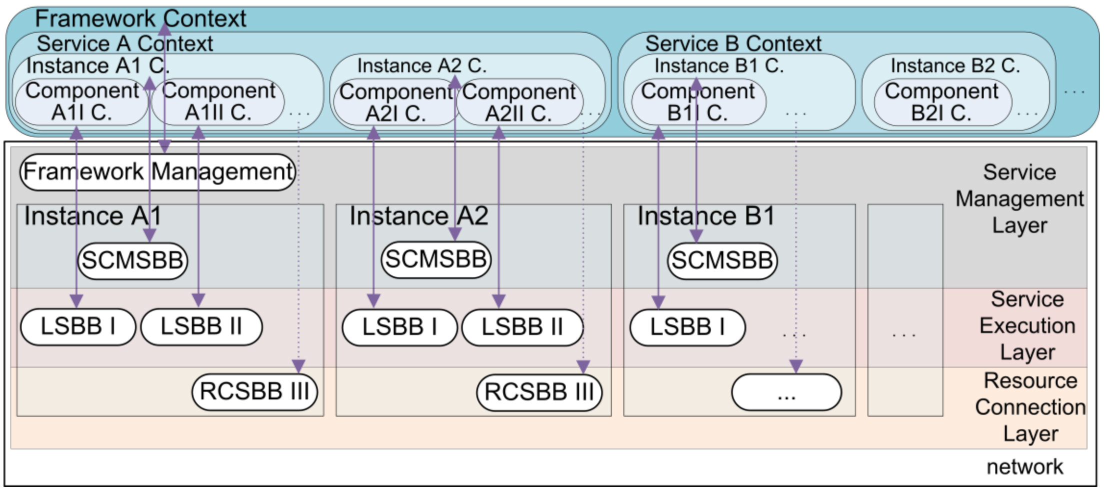


Figure 7.1: Framework context

Each service stores its service context with the service description, parameters, and all service instances in its service context. The service instances use the instance context to store references of their service components (SCMSBB, LSBBs, and the RCSBBs) and references of the variable types (refer to section 7.1.2). The service components use the component context to store their information, e.g., references of their variables, the component status information, and the references to the communication channels (refer to section 7.1.4), which are used to communicate with other service elements or with the framework management.

To get access to their contexts, all service components, service instances, services, and the framework have to retain a reference to their contexts. A service instance can access the contexts of components that belong to the instance but not of other service instances. This allows to monitor the status of subordinated components and prevents the manipulation of a service instance context from another service instance.

7.1.2 Variables and Variable Types

Like other programming languages, BPEL (refer to section 3.6) (OASIS, 2007) uses variables to hold temporary values. Different variables are supported, e.g., WSDL Message type for web service messages, XML Schema type for simple, or complex XML Schema types, XML Schema element for the element attributes, and Build-in type variables for standard and simple types.

The BPEL variables are analysed by the service description parser and mapped to their representations in the SEE (refer to section 6.2.1). The SEE has to implement the variable types that are defined by BPEL.

Whenever a new variable instance is created, this instance is associated with the service instance context (Figure 7.2). If the variable is defined as a BPEL global variable, then the variable instance can be accessed from all SBBs that belong to the same instance of the service. If the variable is defined within a BPEL scope, then it can only be accessed from SBBs belonging to the same scope. This ensures that SBBs can only access variables that are defined in the same scope or globally. Each service instance defines its own set of variables. LSBBs and RCSBBs of one instance cannot access variables of another instance.

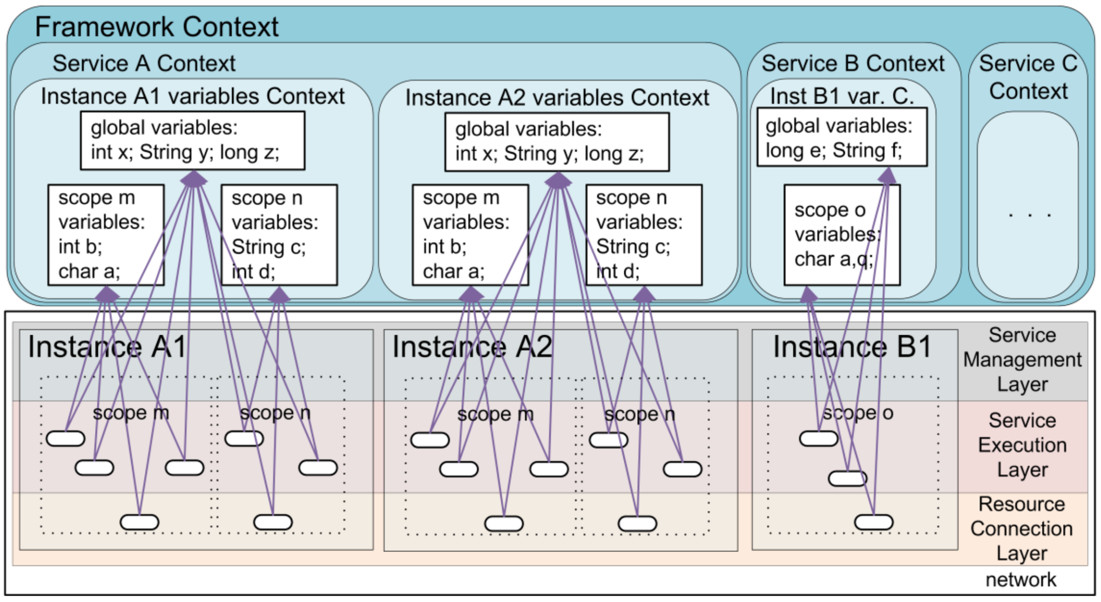


Figure 7.2: Variable context

Some resources and functionalities may require special variable types, e.g., complex variable types, which are not available in BPEL by default. In this case, the CBB on the one hand has to provide a WSDL/XSL document where the variable types are defined to make them available in BPEL and on the other hand, it has to implement the variable types to make them available for the services within the SEE.

7.1.3 External Service Communication

The SEE of the proposed framework is based on the JAIN SLEE specification 1.1 (JSR 240, 2008). The access to the SEE needs to be conformant with the JAIN SLEE specification. The possibilities to send and to receive events to and from JAIN SLEE are restricted in order to ensure the consistence of the services.

All SBBs and, therefore, MSBBs, LSBBs, and RCSBBs have to comply with the specified rules. They all communicate with the help of events. The SBBs are triggered by events and send events to other SBBs. For the service communication within the framework and the service instances, the novel concept of communication channels is proposed for the framework. This concept is described in the subsequent section 7.1.4.

To communicate with other services or to receive and send events between the SLEE and other services, e.g., a web application, two possibilities are defined in the specification.

The first possibility is the standard way for communication with the SLEE using RAs. RAs are able to listen on the network interfaces for external protocol messages. If such a protocol specific message is received, the RA generates an event and fires this event to the event router (refer to section 4.4). RAs can also be called from the SBBs to generate protocol-specific messages and send them to the network. External protocol messages can be, for example, a SIP INVITE message for establishing a communication session but also reports from a temperature sensor or a HTTP request from a web browser.

The second possibility to communicate with the SLEE is to use EJBs. The JAIN SLEE specification describes a method to exchange events between EJBs and JAIN SLEE. EJBs can use the Java EE Connection Architecture (JCA), specified in (JSR 16, 2000), to communicate with external resources by firing and receiving events from JAIN SLEE.

7.1.4 Communication Channels

For the communication between the framework management SBBs with the service instance SBBs, a novel concept of communication channels has been developed. The communication channels are based on the JAIN SLEE event model (refer to section 4.4) but they define a channel from the source SBB to the destination SBB that is created at configuration time of the service instance and the framework component, respectively. This channel is used for event-based communication. As illustrated in Figure 7.3, a communication channel offers a unidirectional point-to-point communication path between two SBBs.

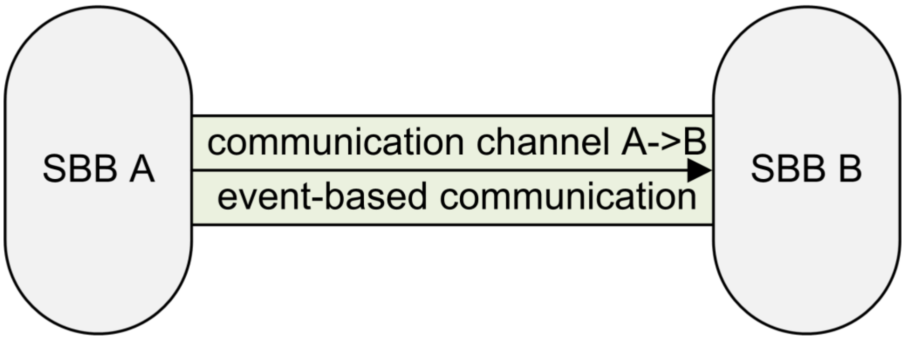


Figure 7.3: Unidirectional communication

To establish a bidirectional communication path, two communication channels have to be defined (Figure 7.4), one from SBB A to SBB B and the other from SBB B to SBB A.

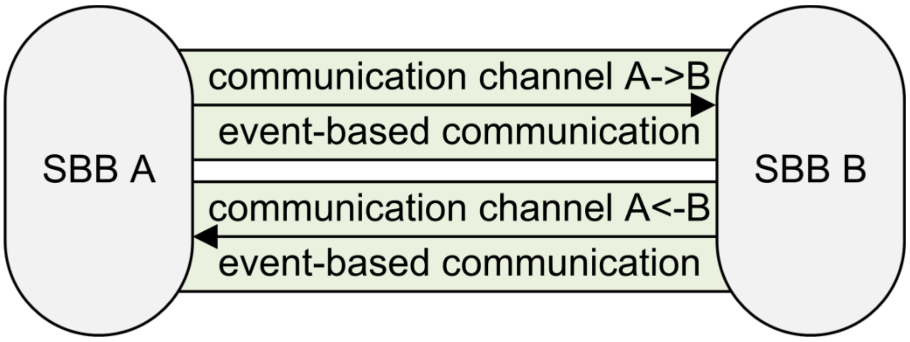

SBB A
communication channel A->B
event-based communication
communication channel A<-B
event-based communication
SBB B

Figure 7.4: Bidirectional communication

The communication channels are stored within the context of each component, are able to fire, and receive a specified type of event. Different types of events have been defined, such as configuration events for signalling and delivering new configuration data, inter service events for the communication between service components, and ready events for signalling that the component or the service is ready for something or that an action is completed.

7.1.5 The Components of a Service

A service can consist of multiple service instances. Each service instance by itself consists of multiple service components. The structure of a service instance is given in Figure 7.5. A service instance consists of one Service Control Management SBB (SCMSBB) controlling its life cycle. The SCMSBB is part of the service management layer (refer to section 6.3.1). To realise the service logic, a service instance also consists of one or multiple Logic SBBs (LSBBs) (refer to section 6.3.2) which belong to the service execution layer. The methods described in the Communication Building Blocks (CBBs) and realising the functionality of the service description are implemented in the Resource Connection SBBs (RCSBBs). These RCSBBs are part of the Resource Connection Layer (refer to section 6.3.3).

The components of a service instance can communicate with each other by exchanging events via the communication channels (refer to section 7.1.4).

The SCMSBB uses communication channels to communicate with the framework management and to control the LSBBs and RCSBBs (Figure 7.6). The framework management fires and sends configuration events to the SCMSBB to trigger the creation, the reconfiguration, or the removal of a service instance (refer to section 7.2.2).

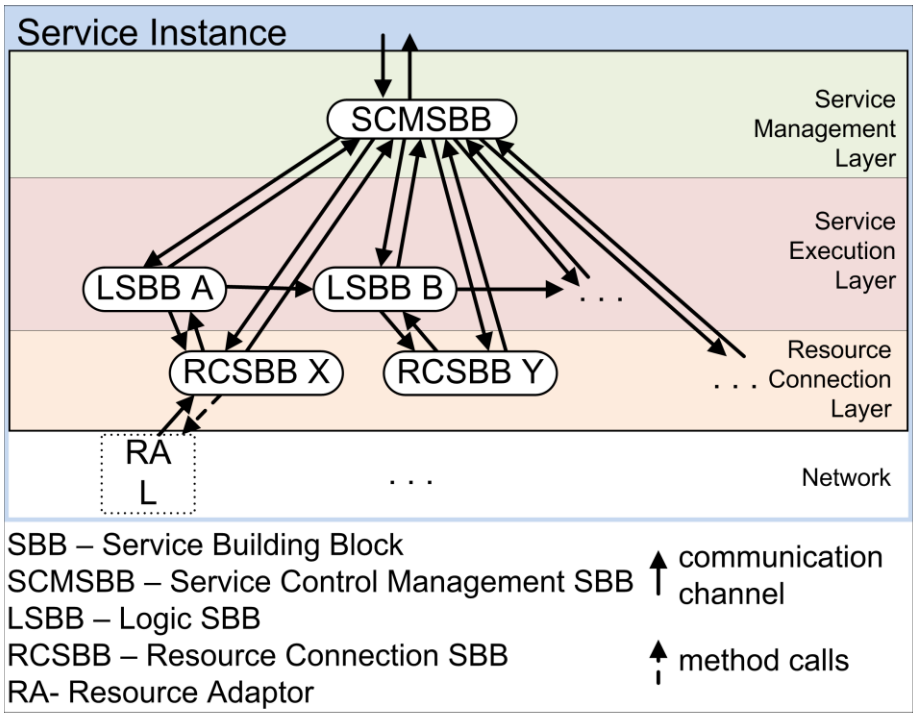


Figure 7.5: Components of a service instance

The SCMSBB sends configuration events to all LSBBs and RCSBBs of the instance and waits until all these SBBs have been created and configured. Upon the SCMSBB has received the ready events from all LSBBs and RCSBBs of the instance, a final ready event is fired to the framework management. In order to trigger the execution of a service instance, the framework management sends an inter-service event to the SCMSBB (refer to section 7.2.3). The SCMSBB then activates the service and sends an inter-service event by itself to the first LSBB in the service. This is always the (main) sequence LSBB that represents the main sequence activity in a BPEL process. After the service workflow has been executed, the last LSBB or RCSBB sends an inter-service event to the SCMSBB to indicate that the service execution has been completed. This event is also signalled to the framework management with an inter-service event.

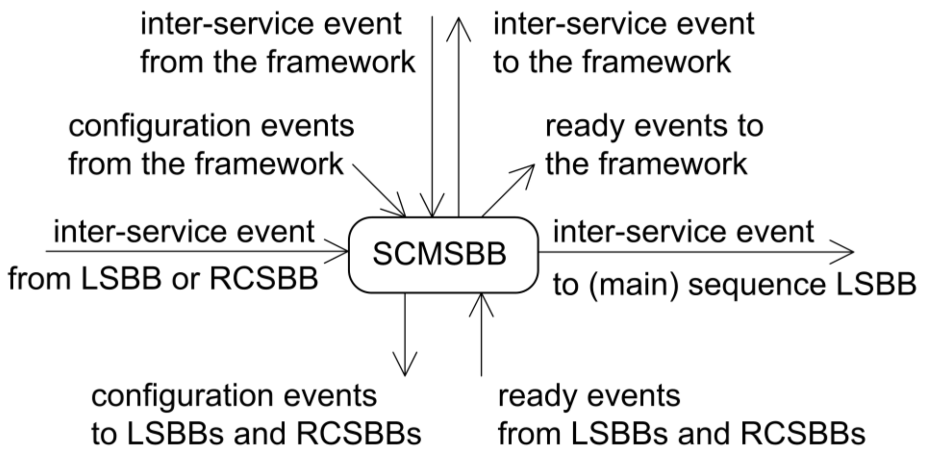

inter-service event from the framework
inter-service event to the framework
configuration events from the framework
ready events to the framework
inter-service event from LSBB or RCSBB
SCMSBB
inter-service event to (main) sequence LSBB
configuration events to LSBBs and RCSBBs
ready events from LSBBs and RCSBBs

Figure 7.6: Communication channels of the SCMSBB

The LSBBs (Figure 7.7) communicate with other LSBBs, RCSBBs, and the SCMSBB to execute the service logic. They receive configuration events from the SCMSBB to create, configure, reconfigure, and remove the LSBB and confirm the completion of these requests with a ready event. The execution of the service logic of the LSBB is triggered by an inter-service event from its predecessor SBB. The LSBB signals the completion of execution to its actual successor SBB using an inter-service event.

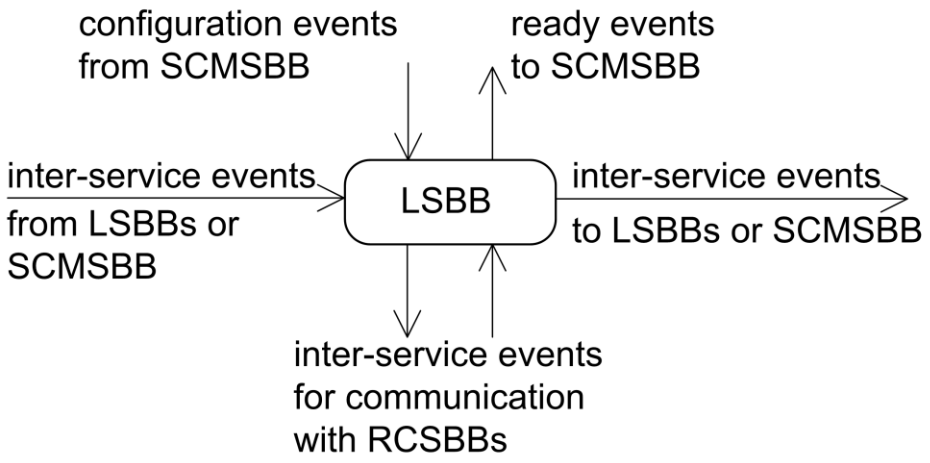

configuration events from SCMSBB
ready events to SCMSBB
inter-service events from LSBBs or SCMSBB
LSBB
inter-service events to LSBBs or SCMSBB
inter-service events for communication with RCSBBs

Figure 7.7: Communication channels of a LSBB

The RCSBBs communicate with the SCMSBB, with LSBBs, and, with components, e.g., RAs that implement the CBB functionality described in the BPEL partner links

(Figure 7.8). They receive configuration events from the SCMSBB. These configuration events can request the creation, configuration, reconfiguration, and the removal of the RCSBB. A ready event is sent to the SCMSBB to confirm the completion of the request.

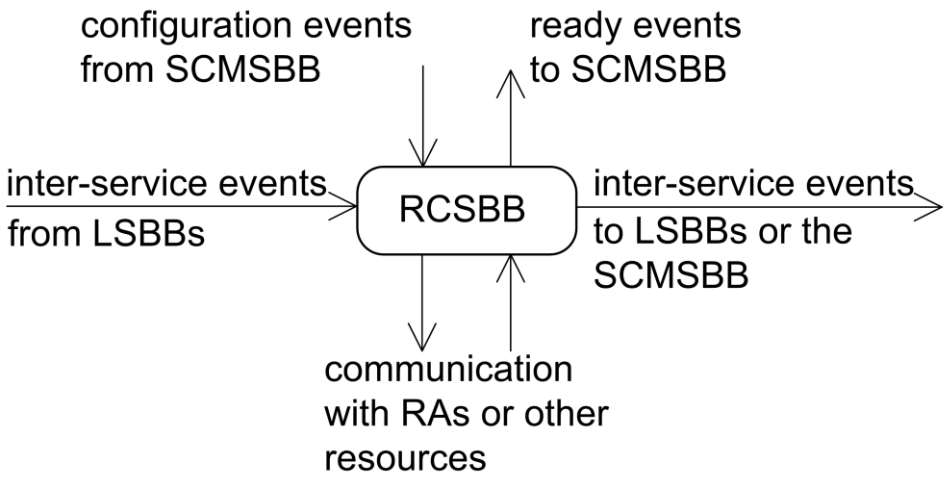

configuration events from SCMSBB
ready events to SCMSBB
inter-service events from LSBBs
RCSBB
inter-service events to LSBBs or the SCMSBB
communication with RAs or other resources

Figure 7.8: Communication channels of a RCSBB

The RCSBBs implement the methods from the CBBs. Before executing a CBB method with the defined parameters, the RCSBB waits for an inter-service event from a LSBB. After its execution, an inter-service event is sent to the successor LSBB to proceed with the service logic execution. The RCSBBs can also call methods on resource adaptors, send events to the resources which implement a functionality, and receive events from the resource adaptors and other resources.

7.2 Service Life Cycle

The service life cycle consists of multiple phases, (i) the service composition phase (refer to section 7.2.2), (ii) the service execution phase (refer to section 7.2.3), (iii) the service reconfiguration phase (refer to section 7.2.4), (iv) and the service removing phase (refer to section 7.2.5). These phases change the states of a service instance. The states of a service instance are described in the following section.

7.2.1 States of a Service Instance

The state machine of a service instance defines several states (Figure 7.9). If the service description is loaded from the repository or from the file system, the service instance will enter the "described" state. This means that the service description of a service exists but the service instance is not created yet. The framework management initiates the service composition phase (refer to section 7.2.2) to create and configure the service instance. After this phase, the service instance is in the "created and configured" state. In this state, the framework management can trigger the reconfiguration phase, the service execution phase, or the service removal phase.

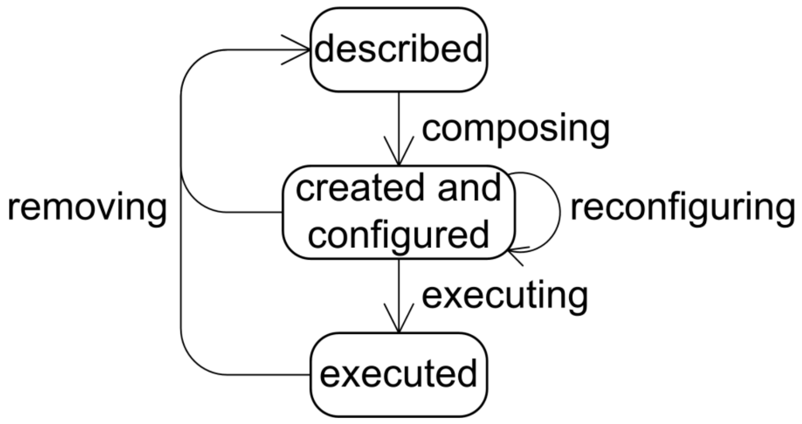


Figure 7.9: States of a service instance

In the reconfiguration phase (refer to section 7.2.4), a reconfiguration of the service instance can be performed. Afterwards, the service is again in the "created and configured" state. If the framework management triggers the service execution phase (refer to section 7.2.3), the service instance will be executed. This will result in a transition to the “executed” state. The framework will remove executed service instances from the memory. Therefore, it will trigger the service-removal phase (refer to section 7.2.5) for this service instance.

Furthermore, service removal phase can be triggered by the framework management in the “created and configured” state, in the “executed” state, in the service composition phase, and in the service execution phase. In the service removal phase the service instance is destroyed, and the resources are freed. After the service instance has been removed, the instance will be again in the “described” state.

7.2.2 Service Composition Phase

In this phase, the service instance is created and configured. This phase requires an existing service description that is available within the service repository. With this service description, the framework management can trigger the service composition phase. A service instance consists of several components (refer to section 7.1.5), the SCMSBB, the LSBBs, RCSBBs, the service context and the variables, and the communication channels. Within this phase, all the components of the service instance have to be instantiated and configured.

The framework management sends events to trigger the creation and configuration of the service components, and upon creation and configuration, the components respond with confirmation events.

For the creation of the service instances, the framework uses the “Orchestration concept” described in section 5.4.3. In this phase, two relevant events, the configuration event, and the ready event are required (refer to section 7.1.5). In the first step, the framework management sends out the configuration events to create new instances of the services or to reconfigure already running service instances. The service management expects a ready event as confirmation of the reception and the execution of the requested tasks triggered by the configuration event.

When the composition of a service has been triggered, e.g., by a web interface of the framework management, the framework management generates a configuration event and adds the references of the requested service description to the event. The event causes the instantiation of a new SCMSBB. The SCMSBB is a component of the framework management layer (refer to section 6.3.1). An SCMSBB is responsible for exactly one service instance. If multiple service instances are created, the framework management has to generate multiple SCMSBBs for the respective service. If an SCMSBB receives a configuration event with the order to generate a new service instance, the SBB directly starts with the composition of the new service instance (Figure 7.10).

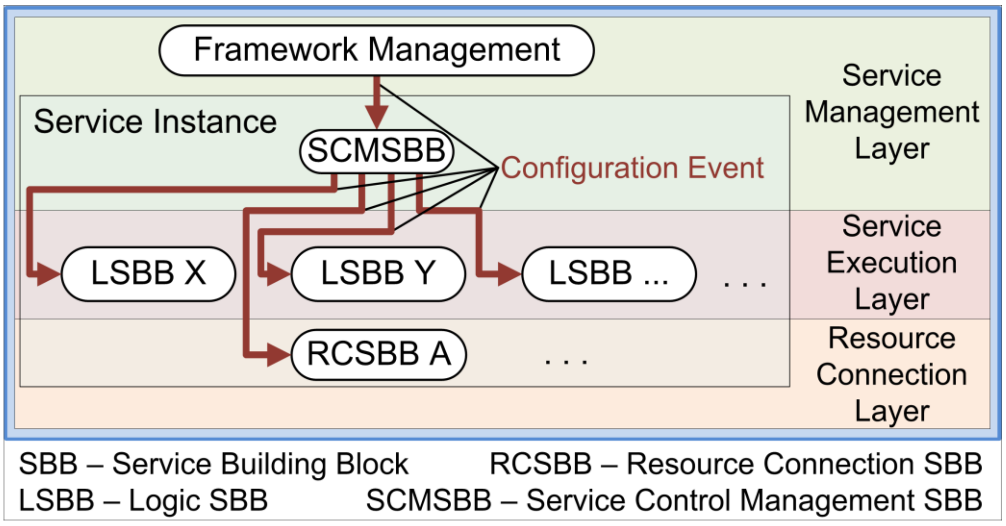


Figure 7.10: Service composition phase – part one

The SCMSBB loads the service description from the repository and starts to parse the description. In the first step, the representation of the instance and the required communication channels are created and stored in the service context (refer to section 7.1.4). In the next step, the required variable types are loaded into the service context. Each service instance has its own context. After this step, the partner links from the service description are parsed, and their CBB representations are referenced in the service context. Furthermore, the description of the variables is analysed. For each variable found, an instance of the variable type is created and referenced in the context. In the next step, the BPEL process description is parsed.

The SCMSBB analyses which service components are required for this service and prepares the service context for the required components. Then the SCMSBB starts sending configuration events to selected components. The components are composed into a new service instance. The events include references to the service context, the framework management, and lists with the required communication channels.

The required communication channels are the ready channels for the communication between the corresponding component and the SCMSBB, and two lists of communication channels for the communication between the service components within the service execution phase.

Furthermore, a communication channel for configuration events is established. This channel is required to send reconfiguration requests to the service component. Each service component gets a unique ID to distinguish the component from other components of the service and from the components of other instances.

The SCMSBB sends the configuration events and expects an answer for each of these events. Each component that has received a configuration event has to respond with a ready event back to the SCMSBB. The SCMSBB sends these configuration events to LSBBs within the service execution layer (refer to section 6.3.2) or to RCSBBs within the resource connection layer (refer to section 6.3.3). Existing components can be reconfigured or deleted with this configuration event. Non-existing components are instantiated, and the configuration event is forwarded to these newly created components.

The basic configuration steps are the same for RCSBB and LSBBs. A RCSBB implements the corresponding CBBs, which allow the service to access the required resources. A RCSBB may implement the resources by itself; call a resource that is offered by the framework, or it may call a resource adaptor to offer the service access to the resource. Which of these possibilities are used by the RCSBB depends on the particular CBB and on the methods defined in the service description.

A RCSBB or LSBB can access the service context with the component ID that was sent with the configuration event, and it can parse its part of the service description. The RCSBBs and LSBBs load and parse their variables; they register themselves to the configuration and ready channels to communicate with their SCMSBB.

Furthermore, the RCSBBs and LSBBs have to register for the inter-service events. With these events, the components communicate with other components during the service execution phase. The configuration event contains references to two lists created by the SCMSBBs and stored in the service context. Both of the lists contain references to channels that are defined for inter-service events (refer to section 7.1.4).

The first list contains references to communication channels of all RCSBBs and LSBBs. These communication channels are required for sending events during the service execution phase. The second list contains references to communication channels that wait for events from this component during the service execution phase.

In the last step, the component confirms completion of successful configuration by sending a ready event back via the ready channel to the SCMSBB (Figure 7.11).

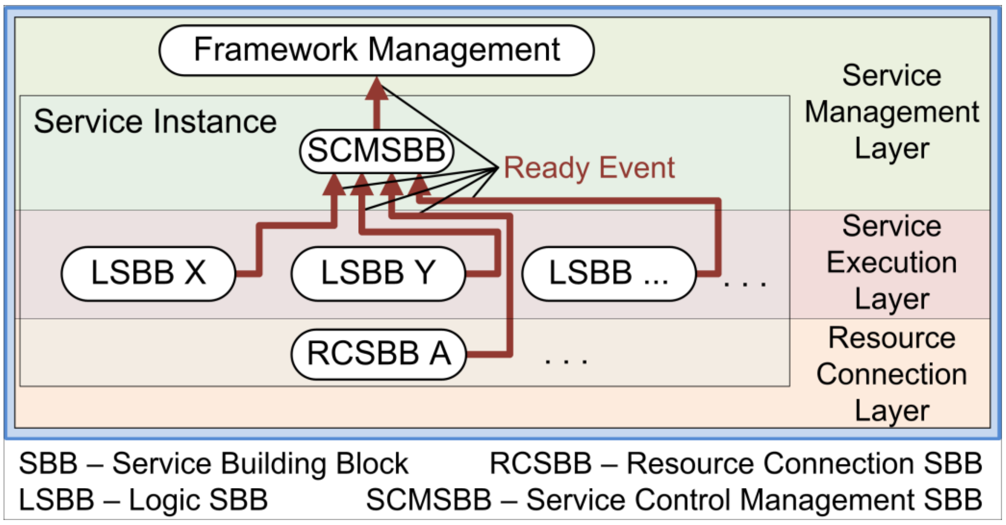


Figure 7.11: Service composition phase – part two

These configuration steps are executed for all RCSBBs and LSBBs. Upon the SCMSBB has received all ready events of all depending RCSBBs and LSBBs, the SCMSBB also generates a ready event and sends it to the framework management to indicate the successful creation and configuration of a new instance. Now, the service instance enters the "created and configured state" (refer to section 7.2.1) and it can be triggered by the framework management for execution.

For a better insight into service composition, a "notification service" example is presented. The service waits for incoming e-mails, which are received by one

RCSBB, and sent out a SIP instant message by another RCSBB. The configuration events are shown in Figure 7.12 and the ready events in Figure 7.13. This service generates and sends a SIP instant message if an e-mail is received. As input for the service creation, the "notification service" description is required. The framework management copies this description for the service composition from the repository, generates a configuration event which includes this description, and sends the event to a newly instantiated SCMSBB.

The SCMSBB receives the configuration event, parses the "notification service" description, and starts with the composition of the service instance.

For each BPEL activity found and for each CBB contained within the description, the SCMSBB generates a configuration event for the corresponding LSBB or RCSBB. The structure of the generated service is derived from the structure of the service description.

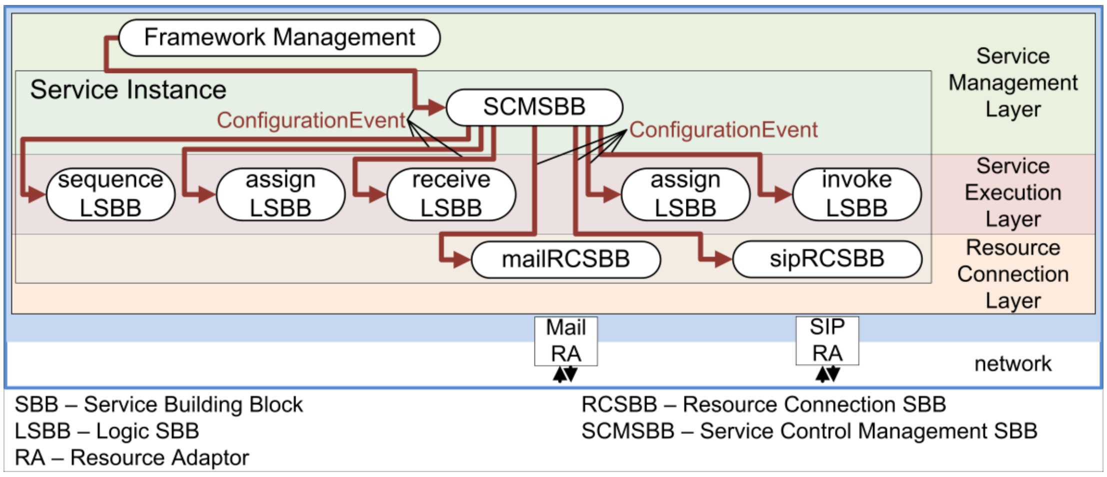
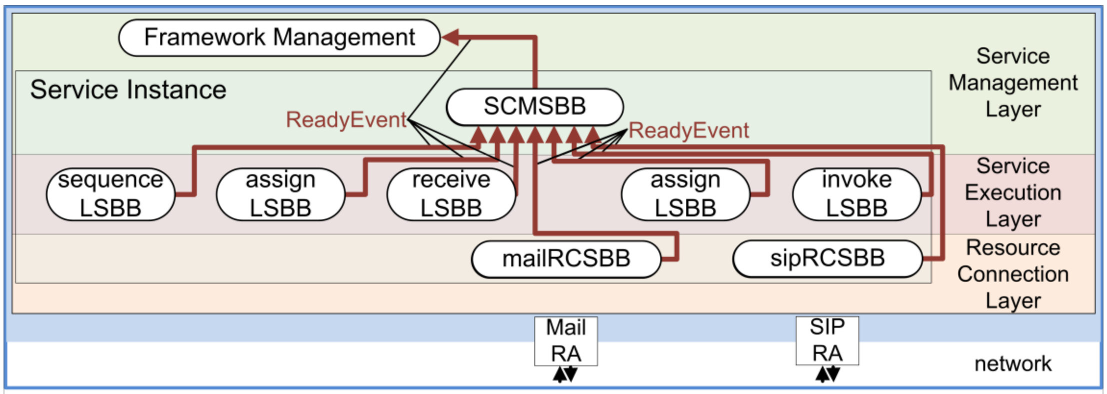


Figure 7.12: Notification example, service composition phase – part one

The notification service consists of five LSBBs and two RCSBBs. The first element is the main sequence LSBB. It holds a list of sub-elements that should be executed sequentially during the service execution phase. This sequence LSBB includes four sub-components, two assign LSBBs, one receive LSBB, and one invoke LSBB. The first element within the sequence is an assign LSBB, followed by a receive LSBB and another assign LSBB. The last element of the sequence is the invoke LSBB. All of the RCSBBs and LSBBs receive the configuration events from the SCMSBB.

Upon an element has been completely the configured, it generates a ready event and sends it back to the SCMSBB (Figure 7.13). If all components of the service are configured and the SCMSBB has received ready events from all components of the instance, the SCMSBB also generates a ready event and sends it to the framework management to signal that the service is ready for execution.



SBB – Service Building Block
LSBB – Logic SBB
RA – Resource Adaptor
RCSBB – Resource Connection SBB
SCMSBB – Service Control Management SBB

Figure 7.13: Notification example, service composition phase – part two

7.2.3 Service Execution Phase

After the composition of services in the previous section, here the execution of services is described. The framework management initiates the execution of a service instance. This can be triggered by a user interaction on the management web interface or automatically, after the service instance was created and configured by the framework management itself. To trigger the execution of a service instance, the framework management fires an inter-service event using the inter-service channel to the SCMSBB of the corresponding service instance (Figure 7.14).

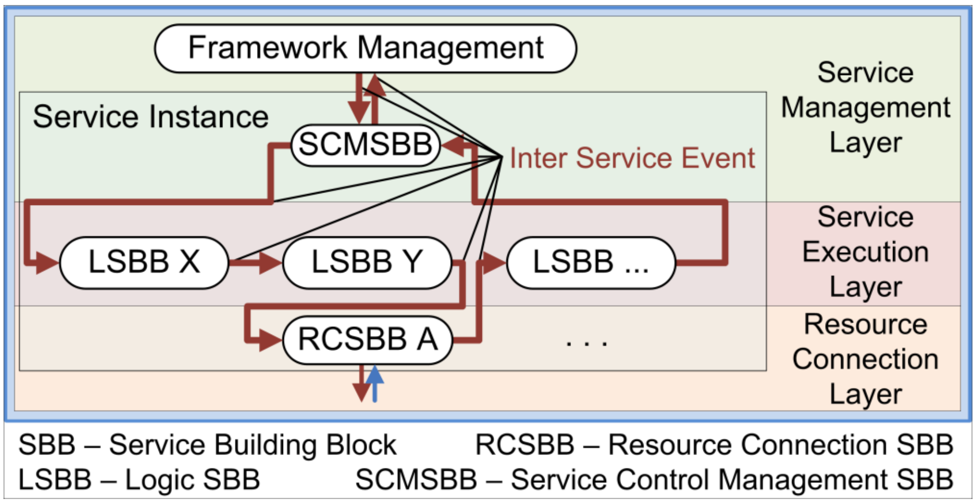


Figure 7.14: Service execution phase

When the SCMSBB receives an inter-service event, the service instance enters the service execution phase. Within this phase, the individual components of the service instance are activated and executed in the order as they are listed in the service description.

The SCMSBB triggers the execution of the service instance by activating its first service component. The inter-service channel of the first service component was stored in the service context during the service composition phase. The SCMSBB loads this channel and sends an inter-service event to the first component. This event serves as service trigger for the execution of the instance.

In the service description, the first process activity is always a BPEL Sequence activity that includes all other BPEL activities within its body. In the service composition phase, an appropriate LSBB has been created that represents this

Sequence activity ([main] Sequence LSBB) (Figure 7.15). So the inter-service event is sent from the SCMSBB to this Sequence LSBB.

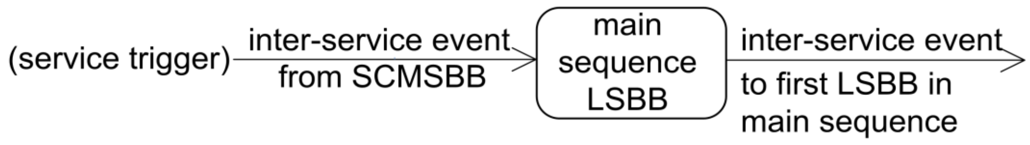

(service trigger)
inter-service event from SCMSBB
main sequence LSBB
inter-service event to first LSBB in main sequence

Figure 7.15 : Triggering the service execution

The Sequence LSBB identifies the next LSBB or RCSBB of the workflow by loading the corresponding inter-service communication channel from the context. During the service composition phase, the communication channels for all service components were established and stored within the service context (refer to section 7.2.2). The next component is triggered for execution by sending an inter-service event from the main Sequence LSBB to this component.

This procedure is repeated for all the LSBBs and RCSBBs of the service workflow. The LSBBs execute the service logic, e.g., by manipulating variable values, handling errors, and choosing the next components for execution. If the LSBB has executed its service logic, it generates an inter-service event and sends it via the inter-service channel to the next LSBB or RCSBB.

The RCSBBs start with their tasks, upon they have received an inter service event. They offer the requested method implementations of the CBBs to the service instance. Similar to the LSBBs, also the RCSBBs send an inter-service event to the next LSBB or RCSBB in the workflow.

Upon the last LSBB or RCSBB of the service workflow has completed its execution, the service instance needs to be stopped, the framework to be informed that the execution of the instance is finished, and the memory and other resources have to be released. The last component loads the inter-service destination channel from its context. This communication channel offers a connection from the last workflow component to the SCMSBB. The inter-service event which is sent via this channel triggers the SCMSBB to release the service context in order to remove the service instance with all components (refer to section 7.2.5) and to close the connections to the resources. To notify a successful execution of the service instance, the SCMSBB sends a ready event to the framework management.

The following example presents the service execution phase for the "notification service" which was described in section 7.2.2. The basic procedure of this phase is shown in Figure 7.16. The depicted service waits for an incoming e-mail. If an e-mail is received by the server, a SIP instant message (IM) is generated and sent.

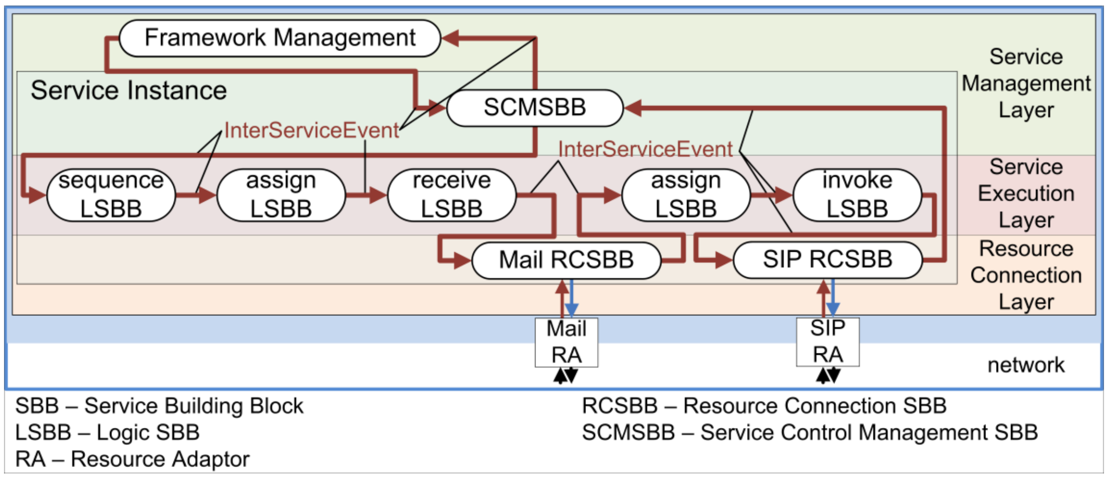


Figure 7.16: Notification example; service execution phase

All involved components are already configured for execution. The framework management triggers the service instance with an inter-service event that is received by the SCMSBB. The SCMSBB fires an inter-service event to the first LSBB of the service instance. As described in section 7.2.2, the first SBB of a service instance is always a sequence LSBB. Within the service instance, all communication between the services elements is performed via inter-service events. The next element in the workflow is the assign LSBB. This LSBB sets the required values of all variables that are necessary for receiving an e-mail. Furthermore, filters for incoming e-mails and outgoing instant messages can be defined. After this step, the receive LSBB is triggered, which prepares the variables and parameters for the Mail RCSBB. The Mail RCSBB is responsible for receiving mails. In this state of the service execution, the Mail RCSBB is activated by the Receive LSBB and waits until it is triggered by the MailRA. The Mail RCSBB is able to receive events from the Mail RA. If the Mail RA receives an e-mail, it generates the corresponding inter-service event. Depending on the defined filter criteria, the Mail RCSBB may be triggered. Upon an appropriate e-mail has been received by the RA, it fires an inter-service event to the Mail RCSBB to trigger the execution of the next component.

The next SBB within the workflow is again an assign LSBB. This LSBB manipulates variables within the service instance. In this example, the destination of the IM needs to be configured, and parts of the content of the received mail need to be copied from the mail content variable to the IM content variable. When all the required variables are set, the invoke LSBB can trigger the SIP RCSBB to send out the IM. The SIP RCSBB is the last service element in the workflow. After the SIP RA has generated and sent out the IM, the SIP RCSBB returns a ready event back to the SCMSBB to signal the completion of the workflow. The SCMSBB will free the resources, clean up the executed service elements, and inform the framework management about the execution of the service instance.

7.2.4 Service Reconfiguration Phase

This section describes the reconfiguration of a service. The framework offers the possibility to modify the configuration and the structure of a service at runtime. It is possible to modify a service that has already been configured or that is being executed.

If the service is configured but not yet executed, all its service instances are in the state “created and configured”. To reconfigure a service, each service instance has to be replaced (Figure 7.17). In case that the execution of the service has already been started, the instances that are in the “service execution phase” are still executing the previous service workflow. However, all service instances that are within or before the “created and configured” state are replaced by the new service instances. After reconfiguration, all new service instances are executing the new service workflow.

For reconfiguration of a service, the previous service description has to be replaced by the new version. This new version of the service description is stored in the service repository and overwrites the previous one. The framework management analyses the running instances of the service. If no service instance of this service is currently being executed, all previous service instances are replaced by the new instances. This procedure is similar to the service composition phase (refer to section 7.2.2), but with the difference that previous service instances will be removed (refer to section 7.2.5).

The framework management sends configuration events to initialise the creation of new service instances. Each previous service instance will be replaced by a new instance. If a new service instance has been generated, the SCMSBB of this new instance confirms the instance creation by sending a ready event to the framework management. If the framework management receives such an event, it will start to remove the previous instance. To do so, the framework management generates a new configuration event and sends it via the configuration channel to the previous SCMSBB to trigger the removal of the instance components. After the previous service instance has been released and all components have been removed, the SCMSBB confirms this step with a ready event. The framework management repeats this procedure for all service instances that need to be replaced. Finally, all service instances of the service represent the new workflow. When the first new service instance reaches the state "created and configured", the service can be executed. The framework management does not need to wait until all instances are reconfigured. Each new service instance can be executed upon reconfiguration.

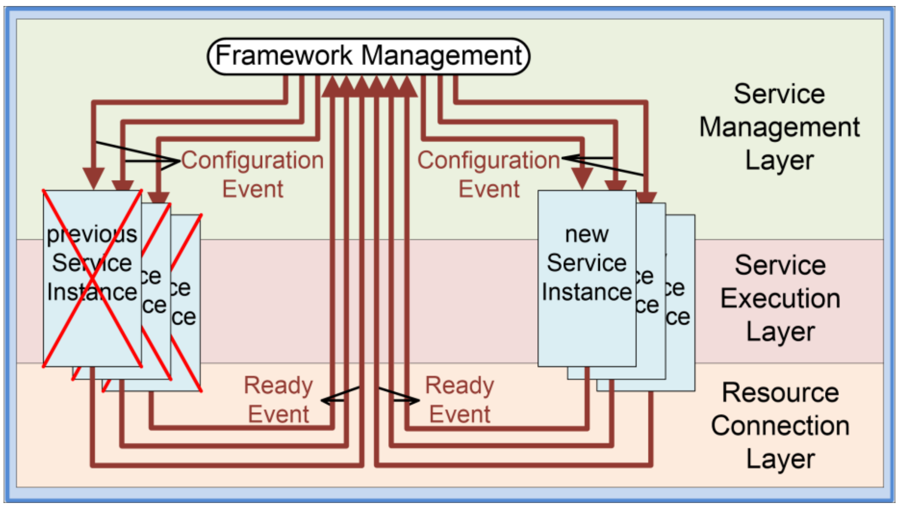


Figure 7.17: Reconfiguration by replacing previous instances

In case that a service should be replaced by a new version of the service and some of the current service instances are already being executed, then the procedure needs some modifications.

The service instances that are in the “created and configured” state will be replaced by instances of the new service as described before. Previous service instances will execute the remaining workflow using the previous configuration. After the SCMSBB of a previous service instance confirms the execution and removal of its service instance by firing an inter-service event to the framework management, the new service instance is created from the new version of the service description. With this strategy, all previous versions of the service instances are replaced by new versions, as soon as the SCMSBB of the previous service instance confirms the completed execution of its instance.

7.2.5 Service Removal Phase

The framework management or the SCMSBB can decide to remove a service instance.

The framework management triggers the service removal phase to remove a service instance from the SEE. This can be initiated, for example, using the web interface or during the reconfiguration phase. If a complete service together with all its instances shall be removed from the SEE, the framework management triggers the service removal phase for all service instances of this service.

The SCMSBB triggers the service removal procedure after the service execution phase to remove all remaining LSBBs and RCSBBs of its service instance. The LSBBs and RCSBBs that have been used during the service execution phase will automatically be removed after their execution has been completed. However, components of a service instance that have not been executed during the service execution phase, e.g., these components were part of a branch that was not executed, have to be removed with the following procedure after the execution of the service instance.

The service removal phase is triggered by a configuration event. In case that the framework management is the initiator of this phase, it sends a configuration event that contains the removal instruction to the SCMSBB (Figure 7.18).

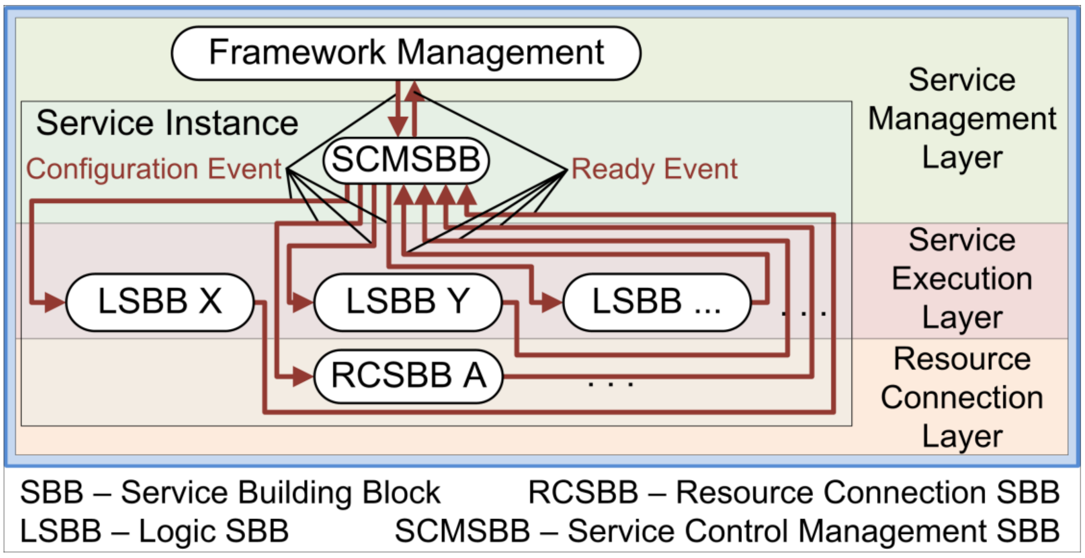


Figure 7.18: Service removal phase

The SCMSBB analyses this event and sends configuration events that contain the removal instruction to all components of the service instance. These components parse the configuration event and release their resources. To confirm the execution of the removal instruction, they send a ready event to the SCMSBB.

The SCMSBB waits for the ready events from the service components and releases its own resources allocated in the SEE. It also removes its service context and confirms the removal to the framework management by sending a ready event.

7.3 Conclusion

This chapter described the concept of the services, their structure, and life cycle. The proposed novel service structure enables the mapping of the BPEL service description to the JAIN SLEE architecture. The service structure that is automatically generated from the service description consists of multiple service components (refer to section 7.1.5).

One central SCMSBB is responsible for each service instance. This has the advantage that all components can be controlled and monitored in this central place. During the service execution phase (refer to section 7.2.3), the service components are communicating directly with each other and the central element is only required for triggering the execution. This novel service concept combines the advantages of the “Orchestration concept” (refer to section 5.4.3) and the “Choreography concept” (refer to section 5.4.4). The “Orchestration concept” supports the service configuration, service control, and service monitoring. The “Choreography concept” in turn supports the direct communication between the service components. Each component executes its part of the service and delegates the remaining parts to the subsequent components.

The context concept (refer to section 7.1.1) introduced in this chapter offers a location to store all the component-specific configuration information and a possibility to access the variable values in a consistent and centralised way. It allows the monitoring of the framework and of the service instances.

With the “Variables and Variable Types” concept (refer to section 7.1.2), it is possible to map the BPEL variable types within the service description to the corresponding implementations of the SEE. New variable types can be part of CBBs. These CBBs have to include the WSDL description and an implementation of the variable types.

Section 7.1.3 describes two possibilities for RCSBBs to communicate with external resources. The normal way is the use of RAs for external protocol communication. Another possibility is an event-based communication with EJBs. Both possibilities are described in the JAIN SLEE specification (JSR 240, 2008) and are allowed for RCSBBs.

The new concept of communication channels (refer to section 7.1.4) was introduced for the internal service communication. This concept offers an event-based communication between the components within a service instance and enables the framework to communicate with the service instances.

The second part of the chapter describes the service life cycle (refer to section 7.2). This life cycle consists of the states “described”, “created”, “executed”, and of the phases “composition”, “removal”, “execution”, and “reconfiguration”.

Each RCSBB and LSBB has its own configuration. This configuration is performed during the composition phase. The BPEL activity or CBB, which is represented by a SBB, is derived from the BPEL process. Depending on the service, more than one instance of a LSBB or a RCSBB may be necessary. The configuration is set at runtime and can be changed in the reconfiguration phase.

The advantage of this concept is that only one Deployable Unit (DU) (refer to section 4.4) has to be compiled and deployed for each type of a LSBB and a RCSBB. The LSBBs and RCSBBs that are required for a service do not need to be deployed for each service, they just need to be deployed once, e.g., while starting the Service Execution Environment. If a RCSBB or LSBB is required in a service instance, an instance of the component with the specific configuration is created and initiated during the composition phase.

8 Framework and Prototype Evaluation

In this chapter, the framework is evaluated and the research prototype is introduced. Each of the defined criteria (refer to section 5.1) is analysed whether it fulfils a specific criteria within the framework (refer to section 8.1). Afterwards the research prototype, which was used for the proof of concept of the framework, is introduced. The research prototype was designed to demonstrate the overall framework functionality. It consists of the important components of the SCE and SEE to allow the creation of value-added multimedia services. The architecture of the prototype is introduced in section 8.2. In section 8.3, the proof of concept of the SEE components is demonstrated. The qualitative analysis of the requirements for the proposed framework concept is described in section 8.4. An analysis of the quantitative requirements established in this PhD thesis and a comparison between conventional service development and the service description with the PhD framework prototype is presented in section 8.5. There, an analysis of the framework performance and scalability is carried out, too.

8.1 Evaluation of the Defined Framework Requirements

This section evaluates the framework regarding its requirements as defined in section 5.1. Each of the six requirements is analysed whether it is fulfilled within the framework.

- The first requirement was an automated solution that supports the description, creation, execution, and provisioning of value-added telecommunication services. To fulfil this requirement, the developed framework supports all of these parts, the description, the creation, execution, and the provisioning of the service. The service is described in BPEL (refer to section 5.2 and section 6.2), which can be done with a graphical/text-based BPEL development tool. The service description parser (refer to section 5.3.2) analyses the service description, and triggers the creation of the service. The service is composed automatically from pre-defined SBBs (refer to section 7.2.2). After a successful creation and configuration of the service, it can be executed, provided and managed by the SEE (refer to section 6.3).
- The service development should support a graphical method for describing the services. The framework supports BPEL as service description language (refer to section 5.2). Several service development tools are available for BPEL; most of these tools support a graphical process design and, furthermore, a text-based service development. The service designer can choose the method which he prefers for describing the service.
- The developer is able to concentrate on describing the logic of the service. Detailed knowledge of the communication protocols is not necessary; this requirement is solved by the CBBs (refer to section 6.2.1). The CBBs offer, on one hand, a protocol-independent possibility to describe the required functionality of the service in the SCE (partner links) and, on the other hand, the implementation of the functionality within the SEE (RCSBBs, RAs). Moreover, functionalities can support different levels of abstraction. To do

this, the CBB has to offer methods with different granularity for the same functionality (refer to section 5.2.3). Therefore, it is also possible to develop a fine-grained service, which allows a detailed configuration of special aspects of the protocol messages.

- A service consists of several components (SBBs). For these SBBs, a structure is defined that is able to support a broad range of value-added services. The SEE consists of a three-layer structure (refer to section 6.3): a management layer, a service logic layer, and a resource connection layer. The service logic is executed in the service execution layer. Special components, the LSBBs, were developed to execute the service logic in the SEE (refer to section 7.1.5). These LSBBs support the logic functions, which can be described in the SCE with BPEL. The BPEL activities are mapped to these LSBBs (refer to section 5.2.2). Therefore, the developed services support the same logic functions that can be described with BPEL. With the CBBs, it is possible to add new functionalities and protocol support to the services. Therefore, new functionalities and protocols can be supported by the framework by adding them as CBB.
- To describe the service logic, reusable service components (LSBBs) were defined. The service designer describes the service logic with BPEL. The service is generated from the workflow that was described in BPEL. The BPEL activities are mapped to LSBBs in the SEE. For each type of activity in BPEL, a LSBB version exists. The service description parser of the service analyses the service workflow. For each activity in the workflow, it triggers

the creation and configuration of the correspondent LSBB in the SEE. The service is generated from the workflow that was described in BPEL.

- Reusable service components offer the functionality for the value-added services. To define services that use these functionalities, the description of the reusable service components also needs to be available in the service description language. Regarding the reusable service components, a mapping is defined which maps these components from the service description to the SEE. This requirement is solved with the CBBs.

  The SEE should be able to support a wide range of communication protocols. The integration of new protocols should be possible. Here again, the CBBs come into play (refer to section 6.2.1). The CBBs within the SEE consists of the RCSBBs and RAs (refer to section 6.3.3). The RCSBBs implements the methods, which provide the functionalities that are required for the services. The RCSBBs can use the RAs to communicate with the outside world. The RAs offer the protocol-specific communication.

  The support of a new protocol can be established by providing a new CBB. If the new RA and the new RCSBB is deployed within the SEE, the services can use the new functionalities provided by the CBB. New CBBs, RCSBBs, and RAs can, for example, be acquired from the Marketplace (refer to section 6.2.4). On the other side, the CBBs offer the new functionalities in BPEL as partner links. The service designer can use the partner links in BPEL to describe services that use these new functionalities in a service description. The generated service can call the new RCSBBs and RAs during service execution (refer to section 7.2.3).

Fine-grained elements and coarse-grained elements should be available for the service developer when he is describing the services. As already said, the service designer can choose the level of abstraction (refer to section 5.2.3). A CBB can offer coarse-grained and fine-grained methods to call the functionalities. The developer can choose its level of abstraction by calling and configuring the desired methods. A combination of different levels of abstraction is also supported. Therefore, it is also possible to develop a fine-grained service, which allows a detailed configuration of special aspects of the protocol messages.

The framework supports all defined requirements. For the proof of concept, parts of the framework are implemented. The architecture of the prototype is introduced in the next section.

8.2 Architecture of the Research Prototype

A research prototype was developed to show that the proposed process for the creation of value-added services can be provided efficiently in a consistent and automated manner. The implementation of the prototype included several tasks, the implementation of the SCE, the SEE, the CBBs, RCSBBs, LSBBs, and the MSBB. For the proof of concept of the proposed framework, an example use case was selected and described by using a BPEL design tool. This example service was created and executed by using the prototype framework, and the results were analysed. Not all elements of the proposed framework have been implemented for the proof of concept. The implemented architecture of the proposed framework consists of the elements shown in Figure 8.1.

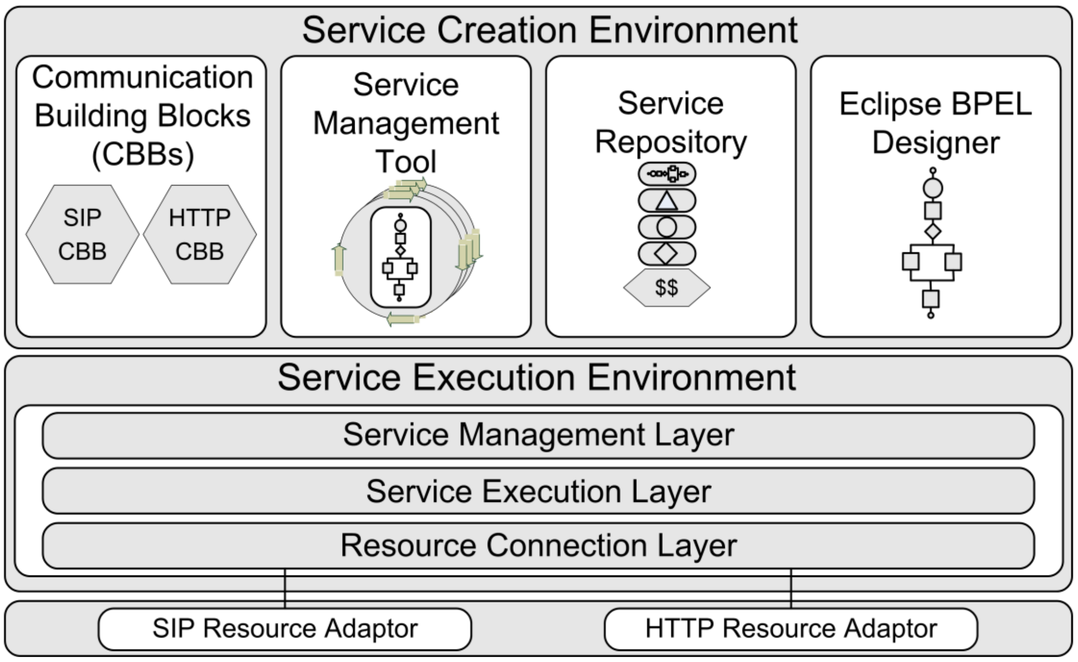


Figure 8.1: Prototype architecture overview

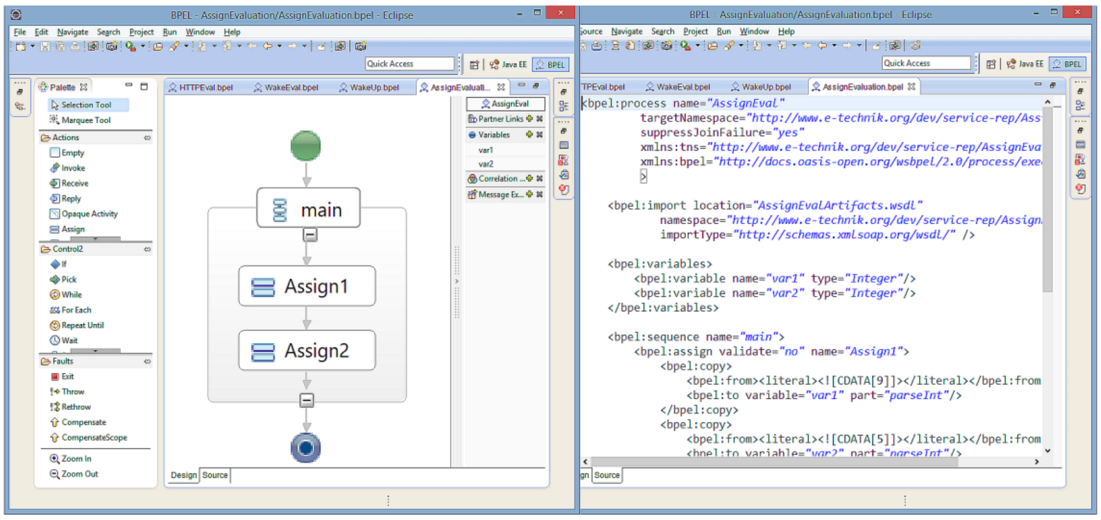
The required parts of the SCE are the CBBs, the service management tool, the service repository, and the graphical service description tool. For the evaluation, two CBBs have been developed, the HTTP CBB and the SIP CBB. The HTTP CBB consists of the HTTP RA, the HTTP RCSBB and the “HTTPServices” BPEL partner link. The SIP CBB consists of the SIP RA, the SIP RCSBB and the “SIPServices” BPEL partner link.

The implemented service management tool offers the possibilities of loading a service description into the framework, monitoring the service, and triggering the service creation, execution, removal, and its stopping.

A repository is used to save the service descriptions. The framework management handles this repository. With the service management tool, new service descriptions can be loaded into the repository, removed from the repository, and loaded from the repository for service creation.

The marketplace is part of the proposed real-world framework architecture (refer to section 6.1), however it has not been implemented in the prototype. It is required for a real-world implementation but not for evaluation purposes. Therefore, the marketplace was not implemented into the prototype.

For service description, the Eclipse BPEL Designer (Eclipse, 2013) was selected. The example service for the framework evaluation is generated with this BPEL tool. The Eclipse BPEL developer tool allows a graphical development with drag and drop features for the service components and the possibility to modify the BPEL document within an XML editor directly. Figure 8.2 gives an impression of the Eclipse BPEL Designer with the graphical drag-and-drop editor on the left and with the XML editor on the right side.



```
<bpel:process name="AssignEval"
        targetNamespace="http://www.e-technik.org/dev/service-rep/Ass
        suppressJoinFailure="yes"
        xmlns:tns="http://www.e-technik.org/dev/service-rep/AssignEva
        xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/exe
        >

    <bpel:import location="AssignEvalArtifacts.wsdl"
            namespace="http://www.e-technik.org/dev/service-rep/Assign
            importType="http://schemas.xmlsoap.org/wsdl/" />

    <bpel:variables>
        <bpel:variable name="var1" type="Integer"/>
        <bpel:variable name="var2" type="Integer"/>
    </bpel:variables>

    <bpel:sequence name="main">
        <bpel:assign validate="no" name="Assign1">
            <bpel:copy>
                <bpel:from><literal><![CDATA[9]]></literal></bpel:from
                <bpel:to variable="var1" part="parseInt"/>
            </bpel:copy>
            <bpel:copy>
                <bpel:from><literal><![CDATA[5]]></literal></bpel:from
                <bpel:to variable="var2" part="parseInt"/>
```

Figure 8.2: Eclipse BPEL developer

Figure 8.3 presents an overview of the implemented framework prototype components and their interaction.

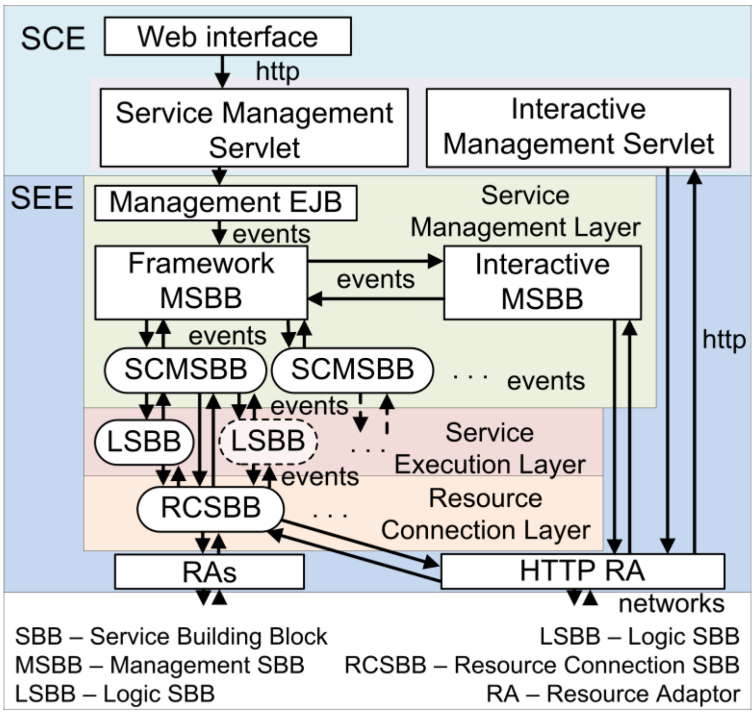
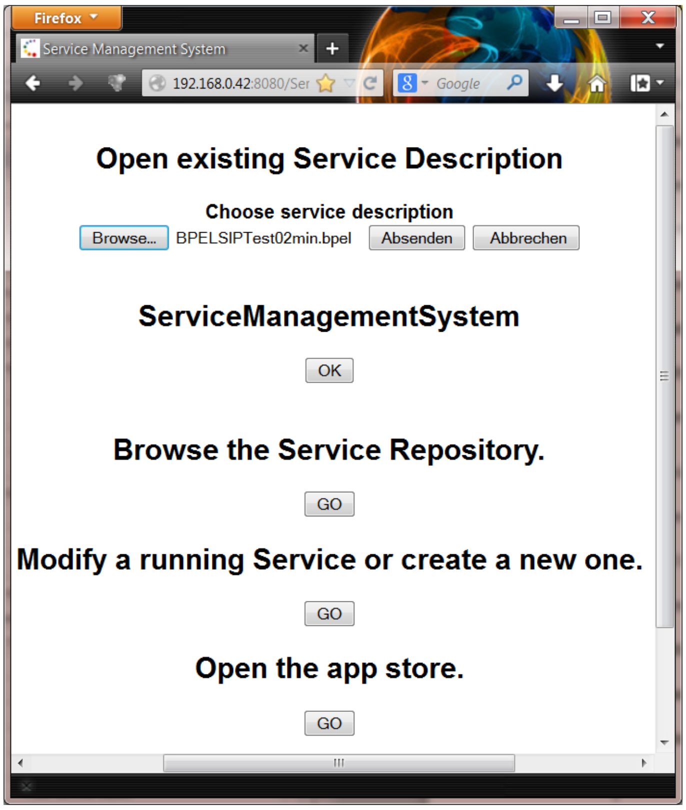


Figure 8.3: Research prototype implementation overview

For loading the service description into the framework repository, a servlet called "Service Management Servlet" in combination with an EJB, called "Management EJB", was developed. A screenshot of the web interface is shown in Figure 8.4. The user of the framework can select the service description, which shall be loaded into the repository. The Service Management Servlet then sends the service description to the Management EJB. With the help of the Java EE Connection Architecture (JCA) (refer to section 7.1.3), the EJB generates a JAIN SLEE event which include the service description and fires this event to the Framework Management. A MSBB, which is called "Framework MSBB", analyses this event and stores the service description in the repository. This MSBB is the main MSBB of the framework. It controls the service life cycle and the framework context.



Open existing Service Description

Choose service description

BPELSIPTest02min.bpel

ServiceManagementSystem

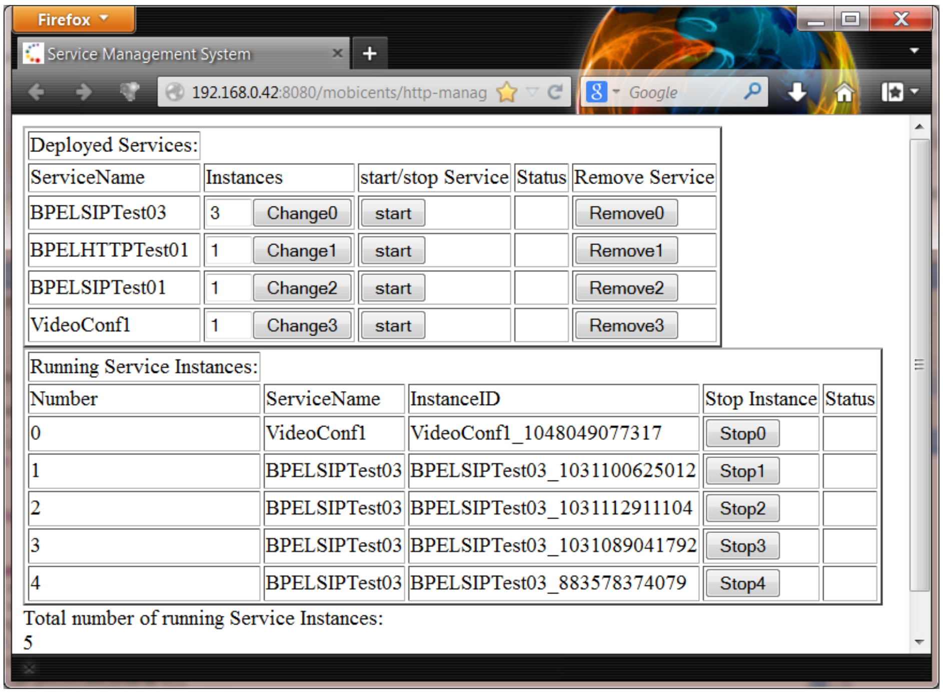
Browse the Service Repository.

Modify a running Service or create a new one.

Open the app store.

Figure 8.4: Service management web interface

The SCE prototype provides another servlet which is responsible for creating, starting, stopping and monitoring the service instances. This servlet is called “Interactive Management Servlet”. The servlet web interface is shown in Figure 8.5. This servlet is communicating with the framework by using a HTTP RA (refer to section 7.1.3). The servlet communicates with the HTTP RA which generates the corresponding JAIN SLEE events and sends them to the framework management. The SEE implementation is based on the open-source Mobicents JAIN SLEE application server (Mobicents, 2014). A MSBB, which is called “Interactive MSBB”, analyses these events and triggers the requested tasks. It manages the monitoring information and offers service control functionalities to the framework user. The Interactive MSBB communicates with the Framework MSBB in order to receive monitoring information from the framework and send requests from the user to the framework. Furthermore, it communicates with the Interactive Management Servlet via the HTTP RA to receive user requests and send monitoring information.


Firefox
Service Management System
192.168.0.42:8080/mobicents/http-manag
Google

Deployed Services:

ServiceName	Instances		start/stop Service	Status	Remove Service
BPELSIPTest03	3	Change0	start		Remove0
BPELHTTPTest01	1	Change1	start		Remove1
BPELSIPTest01	1	Change2	start		Remove2
VideoConf1	1	Change3	start		Remove3

Running Service Instances:

Number	ServiceName	InstanceID	Stop Instance	Status
0	VideoConf1	VideoConf1_1048049077317	Stop0	
1	BPELSIPTest03	BPELSIPTest03_1031100625012	Stop1	
2	BPELSIPTest03	BPELSIPTest03_1031112911104	Stop2	
3	BPELSIPTest03	BPELSIPTest03_1031089041792	Stop3	
4	BPELSIPTest03	BPELSIPTest03_883578374079	Stop4	

Total number of running Service Instances:
5

Figure 8.5: Web interface of interactive management servlet

As described in section 7.2.2, the composition, execution, and removal of service instances is triggered by the Framework Management. In the prototype, the Framework MSBB is responsible for this task. It communicates with the responsible SCMSBB which communicates with the LSBBs of the service logic layer and the RCSBBs of the resource connection layer. For the evaluation of the framework, the SCMSBB and a selection of useful LSBBs and RCSBBs has been developed.

The following components of the service logic layer have been implemented: Assign LSBB, Empty LSBB, Flow LSBB, If LSBB, Invoke LSBB, Receive LSBB, Sequence LSBB, Wait LSBB, and While LSBB.

In the resource connection layer, two RCSBBs have been implemented: The SIP RCSBB offers basic SIP functionality, and the HTTP RCSBB provides HTTP support. As already mentioned, these RCSBBs are part of the corresponding CBBs. The two RAs, which are part of the CBBs, are the SIP RA and the HTTP RA realised by Mobicents JAIN SLEE implementation.

BPEL supports x-Path as standardised scripting language to define expressions (refer to section 5.2.1). For this prototype, the x-Path language was not implemented, but to support conditions for the while- and if- activities and to define expressions within the Assign activity, a limited set of conditions and expressions are supported. For the proof of the concept, the insertion of literals into variables, the reading of variable values, the writing of values into other variables, and the manipulation of a variable with an "`INC`" operation are supported.

Further components have been implemented for the evaluation of the framework: the framework context (refer to section 7.1.1) and the communication channels (refer to section 7.1.5) with ready event, configuration event, inter-service event, and an event for communication with the management EJB interface (refer to section 7.1.3).

8.3 Proof of Concept of the Framework Components

This section describes the implemented components of the framework which are involved in the service instance life cycle. The relevant components are the SCMSBB of the service management layer, the LSBBs from the service logic layer, and the RCSBBs from the resource connection layer.

As already mentioned in the last section, only a limited set of important LSBBs has been implemented for the proof of concept:

- Empty LSBB: This LSBB provides no own functionality. This basic LSBB is used for the proof of concept to evaluate the life cycle phases of the service instances (refer to section 8.3.1).
- Sequence LSBB: Sequential execution of the embedded components is controlled by this LSBB (refer to section 8.3.2).
- Assign LSBB: This LSBB is required for the manipulation of variable values (refer to section 8.3.3).
- Flow LSBB: Parallel execution of components is controlled by this LSBB (refer to section 8.3.4).
- If LSBB: The If LSBB offers the if-condition logic to services (refer to section 8.3.5).
- Invoke LSBB: The invoke LSBB is required to call functionalities of RCSBBs (refer to section 8.3.6).
- Receive LSBB: Waits for events from the RCSBBs (refer to section 8.3.7).
- Wait LSBB: Provides timer functionality to services (refer to section 8.3.8).
- While LSBB: Loop support to services is provided by the while LSBB (refer to section 8.3.9).

In the service life cycle, also the components of the resource connection layer are involved. Two implemented examples, the SIP RCSBB and the HTTP RCSBB, are evaluated in combination with the invoke LSBB (refer to section 8.3.6) and the receive LSBB (refer to section 8.3.7).

8.3.1 Implementation of the Empty LSBB

The Empty LSBB is the most basic component of the service execution layer. It can be used as a representation for the evaluation of other LSBBs. It offers basic LSBB functionality, and it is able to send and receive events. The Empty LSBB can be used to illustrate the service life cycle. In the composition phase (refer to section 7.2.2), the LSBBs will be created and configured, in the reconfiguration phase (refer to section 7.2.4) the configuration of the LSBBs can be modified, and in the removal phase (refer to section 7.2.5) the LSBBs can be deleted. In the execution phase (refer to section 7.2.3), the Empty LSBB waits for an inter-service event from its predecessor. When the LSBB is triggered by this event, it will load the inter-service channel from the context and send an inter-service event to its subsequent component.

Only for the Empty LSBB all four phases are described in this thesis, for all other framework components, only the configuration phase and the execution phase are presented, because the other two phases are almost similar to those shown in case of the Empty LSBB.

For the proof of concept of the Empty LSBB, a BPEL process with one Empty activity was developed. The graphical representation of this process is shown in Figure 8.6. The process consists of a starting point, an end point, and the Empty activity called “Empty”.

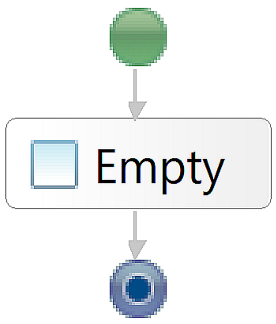
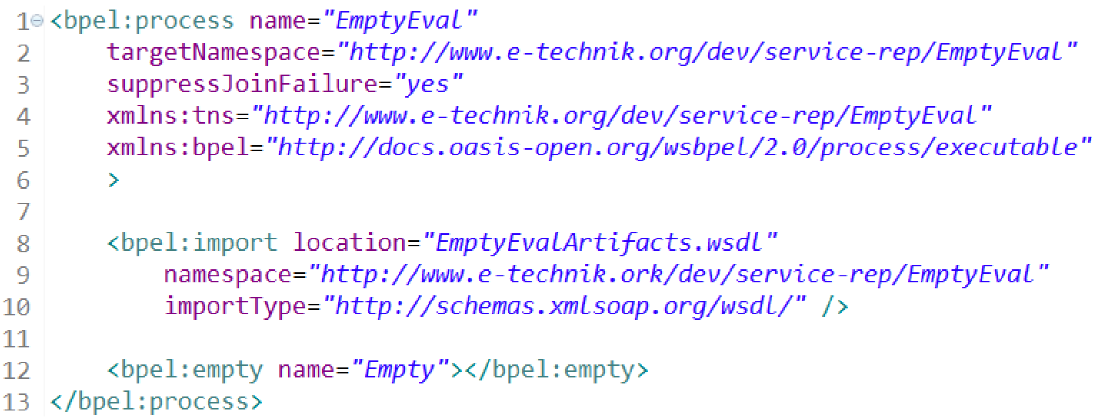


Figure 8.6: Graphical representation of the empty BPEL process

The XML representation of the process is depicted in Figure 8.7. This XML file has been generated using the Eclipse BPEL developer tool. The name of the BPEL process is “EmptyEval” (line 1). A normal BPEL process starts with a Sequence activity that encapsulates other BPEL activities, but for the evaluation, the process parser also allows the Empty activity (line 12). The rest of the process, the namespace definitions (line 2 to 5) and the imports (line 8 to 10), are irrelevant in this case.



```
 1 <bpel:process name="EmptyEval"
 2     targetNamespace="http://www.e-technik.org/dev/service-rep/EmptyEval"
 3     suppressJoinFailure="yes"
 4     xmlns:tns="http://www.e-technik.org/dev/service-rep/EmptyEval"
 5     xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
 6     >
 7
 8     <bpel:import location="EmptyEvalArtifacts.wsdl"
 9         namespace="http://www.e-technik.ork/dev/service-rep/EmptyEval"
10         importType="http://schemas.xmlsoap.org/wsdl/" />
11
12     <bpel:empty name="Empty"></bpel:empty>
13 </bpel:process>
```

Figure 8.7: XML document of the “EmptyEval” process

This BPEL process is uploaded to the service repository, and the life cycle is triggered. The expected behaviour of the framework components in the composition phase is illustrated in Figure 8.8.

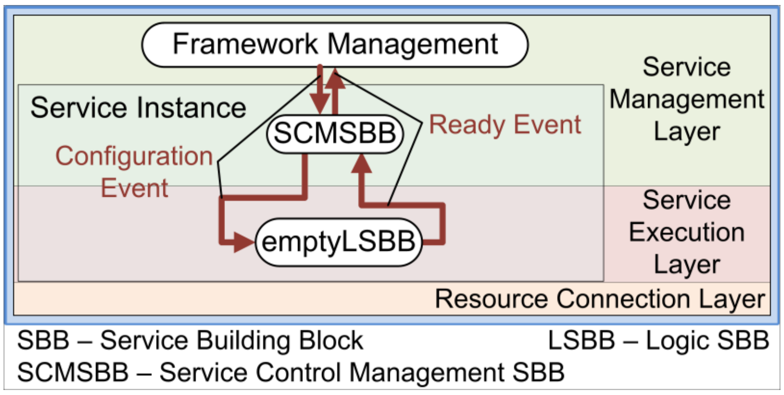


Figure 8.8: Empty composition, reconfiguration, and removal phase

To generate the service, the framework management sends a configuration event to the responsible SCMSBB. This SBB analyses the service description and determines the required components. In this case, only one component, the Empty LSBB, is required to realise the service. The SCMSBB sends a configuration event to the Empty LSBB, and the Empty LSBB sends a ready Event back to the SCMSBB after finishing its configuration steps. The SCMSBB confirms the service creation with a ready event to the framework management, and the service is ready to be triggered for execution. The expected behaviour in the execution phase is shown in Figure 8.9.

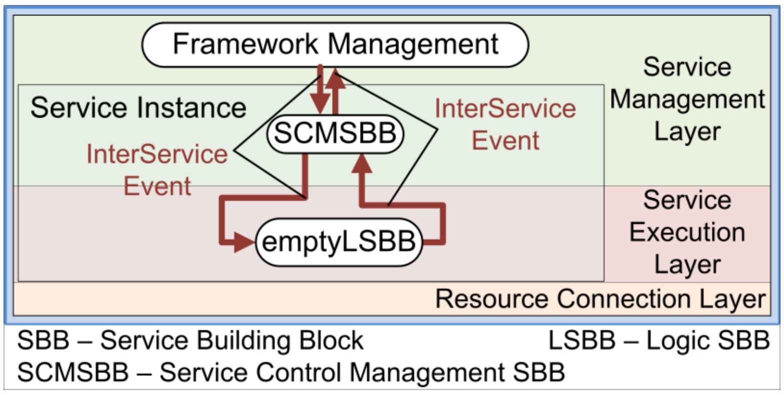


Figure 8.9: Empty execution phase

The framework management triggers the execution phase, it sends an inter-service event to the responsible SCMSBB. This SCMSBB sends an inter-service event to the first LSBB of the service, in this case the Empty LSBB. When the Empty LSBB receives this event, it generates a new inter-service event and sends it to the next service component. This service consists of one LSBB only there is no RCSBB. The Empty LSBB is the first and the last component of the service instance. Because of this, the event is sent back to the SCMSBB. When receiving this event, the service instance is executed and can be removed. The SCMSBB confirms the execution of the instance to the framework management with an inter-service event.

The behaviour in the reconfiguration phase is similar to the behaviour in the composition phase. Instead of setting an initial configuration of the Empty LSBB, an existing configuration is changed to a new one. In the removal phase, the executed Empty LSBB and the SCMSBB will be removed. The received and sent messages will be the same in all three phases, the composition phase, the reconfiguration phase and the removal phase (Figure 8.8).

To evaluate the behaviour of the service components, logging outputs were added to interesting components. Each logging output starts with the name of the component. The framework management is represented as “FrameworkManagementSBB”. The SCMSBB is represented as “ServiceControlMSBB” and the Empty LSBB as “Empty”. In order to give an overview of the exchanged events, screenshots of the logging outputs for all four phases and message sequence charts (MSCs) are shown in the following figures.

The first screenshot in Figure 8.10 represents the logging output of the service composition phase, and Figure 8.11 represents the corresponding MSC.

```
[FrameworkManagementSBB] fire ConfigurationEvent >>
[ServiceControlMSBB] << ConfigurationEvent received
[ServiceControlMSBB] service name:EmptyEval
[ServiceControlMSBB] create new service instance -> parsing description
[ServiceControlMSBB] found imports
[ServiceControlMSBB] found element -> empty -> name: Empty
[ServiceControlMSBB] ElementIdentifier: empty#EmptyEval_10695582926700#1
[ServiceControlMSBB] fire ConfigurationEvent >> empty#EmptyEval_10695582926700#1 >>
[Empty] << ConfigurationEvent received empty#EmptyEval_10695582926700#1
[Empty] fire ReadyEvent-> empty#EmptyEval_10695582926700#1 >>
[ServiceControlMSBB] << ReadyEvent received! << empty#EmptyEval_10695582926700#1
[ServiceControlMSBB] fireReadyEvent >> EmptyEval_10695582926700
[FrameworkManagementSBB] << ReadyEvent received << EmptyEval_10695582926700
[FrameworkManagementSBB] <---> EmptyEval_10695582926700 created and configured <--->
```

Figure 8.10: Empty composition phase

As expected, the framework management triggers the service composition with a configuration event. The SCMSBB receives the event and starts with the analysis of the received BPEL process. The SCMSBB finds the name of the process, the imports, and the Empty activity. To create and configure an Empty LSBB, the SCMSBB sends a configuration event to the LSBB. The Empty LSBB receives the event and returns a ready event when the configuration is finished. The SCMSBB receives this ready event and sends a ready event to the framework management. Now the service is configured and ready for execution.

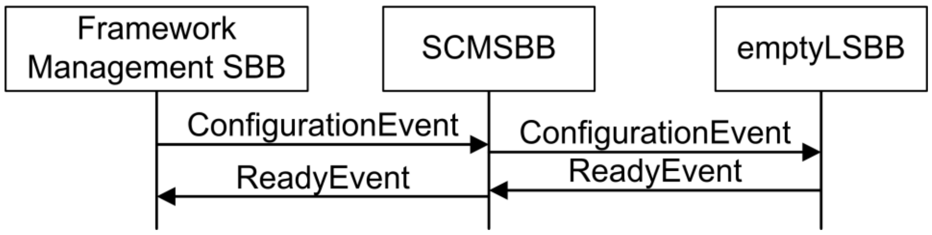
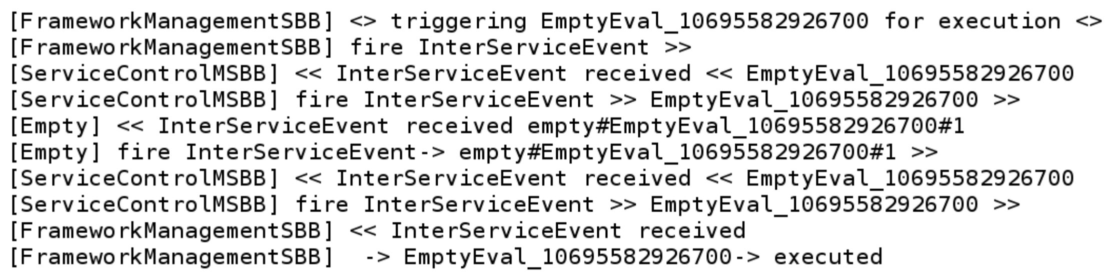


Figure 8.11: Empty composition phase events

The logging output of the service execution phase is shown in Figure 8.12 and the corresponding MSC in Figure 8.13.



```
[FrameworkManagementSBB] <> triggering EmptyEval_10695582926700 for execution <>
[FrameworkManagementSBB] fire InterServiceEvent >>
[ServiceControlMSBB] << InterServiceEvent received << EmptyEval_10695582926700
[ServiceControlMSBB] fire InterServiceEvent >> EmptyEval_10695582926700 >>
[Empty] << InterServiceEvent received empty#EmptyEval_10695582926700#1
[Empty] fire InterServiceEvent-> empty#EmptyEval_10695582926700#1 >>
[ServiceControlMSBB] << InterServiceEvent received << EmptyEval_10695582926700
[ServiceControlMSBB] fire InterServiceEvent >> EmptyEval_10695582926700 >>
[FrameworkManagementSBB] << InterServiceEvent received
[FrameworkManagementSBB]  -> EmptyEval_10695582926700-> executed
```

Figure 8.12: Empty execution phase

The framework management triggers the execution of the service instance with an inter-service event. The responsible SCMSBB receives the event and starts the execution of the Empty LSBB by sending an inter-service event to this LSBB. The Empty LSBB only supports the basic LSBB functionality, which is implemented in all LSBBs, but it has no special extra functionality. The Empty LSBB receives the inter-service event and generates a new inter-service event, which is sent back to the SCMSBB. When the SCMSBB receives this event, the service instance is executed. To inform the framework about the execution of the service instance, an inter-service event is generated and sent to the framework management.

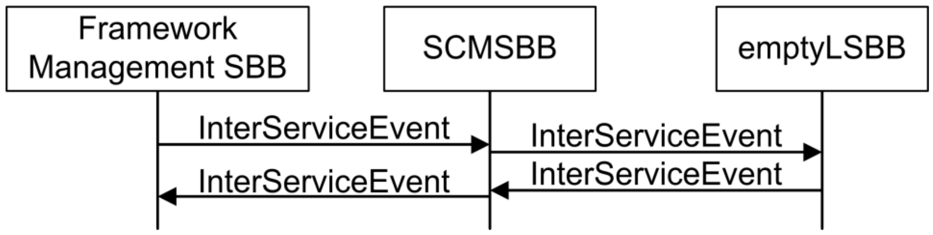
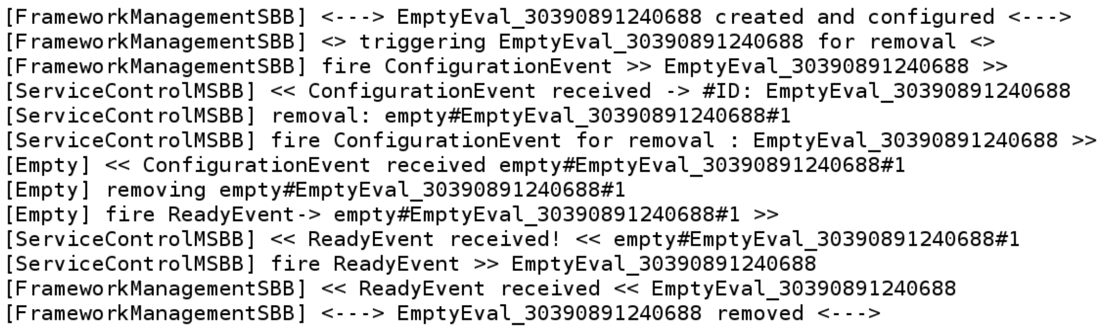


Figure 8.13: Empty execution phase events

The service removal phase is triggered, e.g., when a service instance should be removed from the framework. The screenshot in Figure 8.14 represents the logging output of the service removal phase. The MSC corresponds to the MSC of the service composition phase in Figure 8.11. In this example, the service instance is already created and configured. The framework management triggers this phase with a configuration event. The responsible SCMSBB receives this event and determines all components of the service instance that have to be removed. In this case, the Empty LSBB has to be removed, so the SCMSBB sends a configuration event to this component. The Empty LSBB receives the event and generates a ready event. The ready event is sent to the SCMSBB and the Empty LSBB is removed. The SCMSBB receives this event and waits until all components have confirmed its removal. In this case, only the Empty LSBB has to confirm its removal, so the SCMSBB can generate a ready event and send it as confirmation to the framework management. The SCMSBB removes the service context and the service instance.



```
[FrameworkManagementSBB] <---> EmptyEval_30390891240688 created and configured <--->
[FrameworkManagementSBB] <> triggering EmptyEval_30390891240688 for removal <>
[FrameworkManagementSBB] fire ConfigurationEvent >> EmptyEval_30390891240688 >>
[ServiceControlMSBB] << ConfigurationEvent received -> #ID: EmptyEval_30390891240688
[ServiceControlMSBB] removal: empty#EmptyEval_30390891240688#1
[ServiceControlMSBB] fire ConfigurationEvent for removal : EmptyEval_30390891240688 >>
[Empty] << ConfigurationEvent received empty#EmptyEval_30390891240688#1
[Empty] removing empty#EmptyEval_30390891240688#1
[Empty] fire ReadyEvent-> empty#EmptyEval_30390891240688#1 >>
[ServiceControlMSBB] << ReadyEvent received! << empty#EmptyEval_30390891240688#1
[ServiceControlMSBB] fire ReadyEvent >> EmptyEval_30390891240688
[FrameworkManagementSBB] << ReadyEvent received << EmptyEval_30390891240688
[FrameworkManagementSBB] <---> EmptyEval_30390891240688 removed <--->
```

Figure 8.14: Empty removal phase

The service reconfiguration phase is triggered, e.g., when the configuration of a service instance should be changed. The screenshot in Figure 8.15 represents the logging output of the service removal phase. The MSC corresponds to the MSC of the service composition phase in Figure 8.11. The service instance is already created and configured. The framework management triggers this phase with a configuration event. The SCMSBB of the service instance receives this event and sends configuration events to all components of the instance that need to be changed. In this case, it sends a configuration event to the Empty LSBB. The Empty LSBB changes its configuration and sends a ready event back to the SCMSBB. The

SCMSBB confirms the reconfiguration of the service instance with a ready event to the framework management.

```
[FrameworkManagementSBB] <> triggering EmptyEval_31288386512074for reconfiguration <>
[FrameworkManagementSBB] fire ConfigurationEvent >> EmptyEval_31288386512074 >>
[ServiceControlMSBB] << ConfigurationEvent received -> #ID: EmptyEval_31288386512074
[ServiceControlMSBB] reconfiguration: empty#EmptyEval_31288386512074#1
[ServiceControlMSBB] fire ConfigurationEvent for reconfiguration : EmptyEval_31288386512074 >>
[Empty] << ConfigurationEvent received empty#EmptyEval_31288386512074#1
[Empty] reconfiguring empty#EmptyEval_31288386512074#1
[Empty] fire ReadyEvent-> empty#EmptyEval_31288386512074#1 >>
[ServiceControlMSBB] << ReadyEvent received! << empty#EmptyEval_31288386512074#1
[ServiceControlMSBB] fire ReadyEvent >> EmptyEval_31288386512074
[FrameworkManagementSBB] << ReadyEvent received << EmptyEval_31288386512074
[FrameworkManagementSBB] <---> EmptyEval_31288386512074 reconfigured <--->
```

Figure 8.15: Empty reconfiguration phase

The fundamental functionality of the SCE and SEE has been demonstrated within this section. An example BPEL process that contains the Empty LSBB was analysed. The service components, which are evaluated in the next sections, make use of this functionality.

8.3.2 Implementation of the Sequence LSBB

The Sequence LSBB is mapped to the Sequence activity in BPEL. It is required to describe the sequential execution of activities in BPEL. Most useful service descriptions always require the Sequence activity. Therefore, the sequence LSBB is implemented in the service execution layer.

A Sequence activity is a container for one or more activities that are executed sequentially, in the lexical order in which they appear within the service description of the Sequence activity. A Sequence LSBB is normally the first SBB within a service instance and is triggered by the SCMSBB. In addition, the Sequence LSBB can be encapsulated within other LSBBs to support a sequential execution order within these LSBBs. Examples for this are the While LSBB, the If LSBB, and the

Flow LSBB. The graphical representation of the example BPEL process in Figure 8.16 consists of a Sequence activity called “main” and an Empty activity called “Empty”.

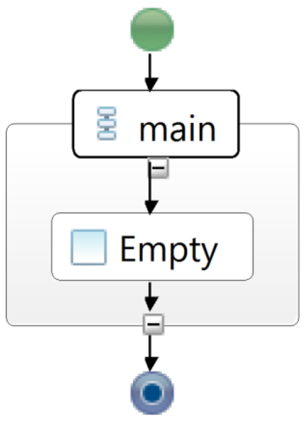
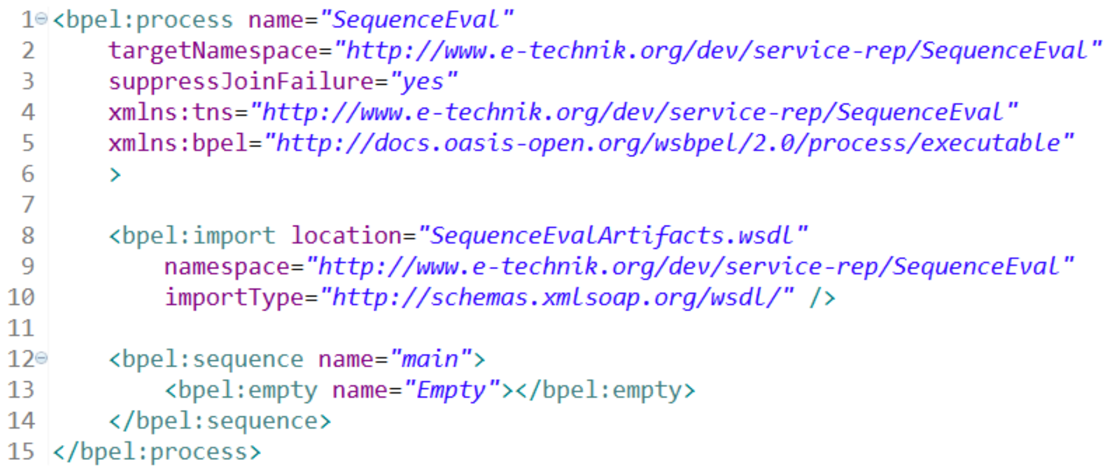


Figure 8.16: Graphical representation of the sequence evaluation BPEL process

The Empty activity is encapsulated within the sequence. The XML document of the BPEL process is shown in Figure 8.17.



```xml
<bpel:process name="SequenceEval"
    targetNamespace="http://www.e-technik.org/dev/service-rep/SequenceEval"
    suppressJoinFailure="yes"
    xmlns:tns="http://www.e-technik.org/dev/service-rep/SequenceEval"
    xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
    >

    <bpel:import location="SequenceEvalArtifacts.wsdl"
        namespace="http://www.e-technik.org/dev/service-rep/SequenceEval"
        importType="http://schemas.xmlsoap.org/wsdl/" />

    <bpel:sequence name="main">
        <bpel:empty name="Empty"></bpel:empty>
    </bpel:sequence>
</bpel:process>
```

Figure 8.17: XML document of the “SequenceEval” process

The BPEL process describes a service that will start executing the sequence. The components within the sequence are executed in the top-down order. The Sequence activity contains only one element, the Empty activity. The name of this BPEL process is “SequenceEval” (line 1). The sequence (line 12 to 14) encapsulates the Empty activity (line 13).

For the evaluation of the Sequence LSBB, the service composition phase and the service execution phase are analysed. In the composition phase (Figure 8.18), it is expected that the framework management triggers the service composition with a configuration event sent to the SCMSBB. The SCMSBB analyses the service description of the BPEL process and configures the required components. The sequence description is analysed for any encapsulated components. Communication channels for the service execution phase are created according to the order of the encapsulated components. The SCMSBB by itself sends out configuration events to create and configure the LSBBs and waits for ready events from these LSBBs. When the ready events are received, the SCMSBB sends a ready event to the framework management to confirm the creation of the service instance.

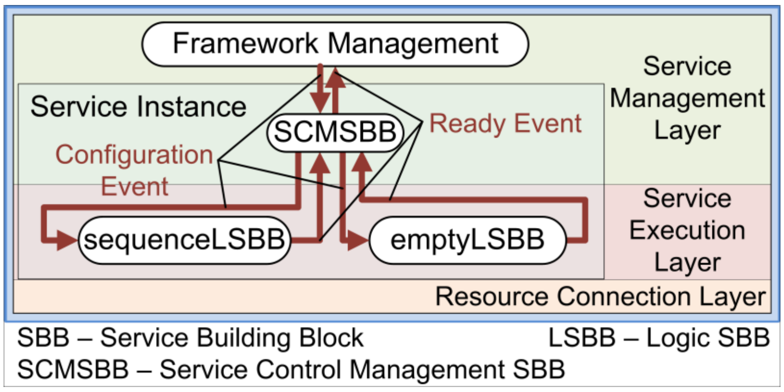


Figure 8.18: Sequence evaluation composition phase

In the service execution phase (Figure 8.19), the framework management triggers the execution of a service instance by sending an inter-service event to the SCMSBB. The SCMSBB starts the execution by sending an inter-service event to the first component of the service, in this case the Sequence LSBB. The Sequence LSBB sends an inter-service event to the first SBB that is encapsulated within the sequence; in this case, the Empty LSBB and the Empty LSBB send an inter-service event back to the SCMSBB. Once the service instance is executed, the SCMSBB confirms the execution to the framework management with an inter-service event.

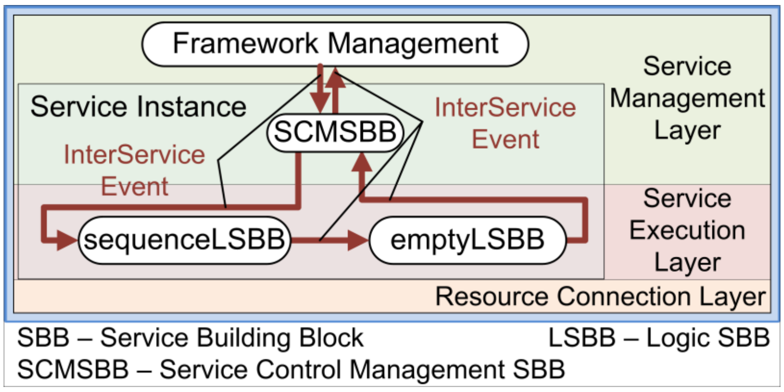


Figure 8.19: Sequence evaluation execution phase

The service description from Figure 8.17 was loaded into the repository and triggered for creation and execution. The log output for the service composition phase is shown in Figure 8.20, and the log output for the service execution phase is depicted in Figure 8.22. For a more detailed analysis, the event communication of both phases is shown in the following MSCs. The MSC for the composition phase is shown in Figure 8.21 and for the execution phase in Figure 8.23.

The log output in Figure 8.20 and the MSC in Figure 8.21 confirm the expected behaviour in the service composition phase. The framework management triggers the creation of a new service instance. The SCMSBB parses the service description and sends configuration events to the Sequence LSBB and to the Empty LSBB. Both LSBBs respond with a ready event and the SCMSBB confirms the creation of the service instance with a ready event to the framework management.

```
[FrameworkManagementSBB] fire ConfigurationEvent >>
[ServiceControlMSBB] << ConfigurationEvent received
[ServiceControlMSBB] service name:SequenceEval
[ServiceControlMSBB] create new service instance -> parsing description
[ServiceControlMSBB] found imports
[ServiceControlMSBB] found element -> sequence -> name: main
[ServiceControlMSBB] ElementIdentifier: sequence#SequenceEval_13804780792980#1
[ServiceControlMSBB] fire ConfigurationEvent >> sequence#SequenceEval_13804780792980#1 >>
[ServiceControlMSBB] found element -> empty -> name: Empty
[ServiceControlMSBB] ElementIdentifier: empty#SequenceEval_13804780792980#3
[ServiceControlMSBB] fire ConfigurationEvent >> empty#SequenceEval_13804780792980#3 >>
[Sequence] << ConfigurationEvent received sequence#SequenceEval_13804780792980#1
[Sequence] fire ReadyEvent >> sequence#SequenceEval_13804780792980#1 >>
[ServiceControlMSBB] << ReadyEvent received! << sequence#SequenceEval_13804780792980#1
[ServiceControlMSBB] Waiting for elements to become ready!
[Empty] << ConfigurationEvent received empty#SequenceEval_13804780792980#3
[Empty] fire ReadyEvent-> empty#SequenceEval_13804780792980#3 >>
[ServiceControlMSBB] << ReadyEvent received! << empty#SequenceEval_13804780792980#3
[ServiceControlMSBB] fire ReadyEvent >> SequenceEval_13804780792980
[FrameworkManagementSBB] << ReadyEvent received << SequenceEval_13804780792980
[FrameworkManagementSBB] <---> SequenceEval_13804780792980 created and configured <--->
```

Figure 8.20: Sequence composition evaluation log

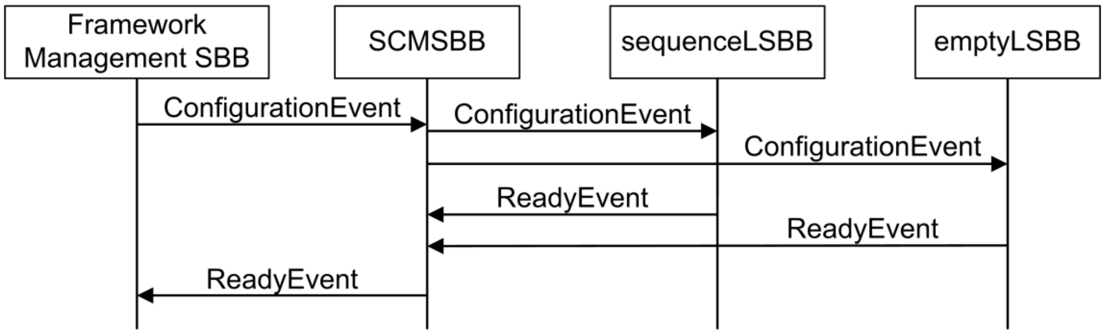


Figure 8.21: Sequence composition phase MSC

The service execution phase (Figure 8.22 and Figure 8.23) is initiated with an inter-service event. This even is send from the framework management to the SCMSBB of the service instance. The SCMSBB fires an inter-service event to the first LSBB of the workflow that is the Sequence LSBB. The next LSBB is the Empty LSBB, encapsulated within the sequence. This LSBB is the last element of the workflow. Therefore, it fires an inter-service event back to the SCMSBB, which confirms the execution of the instance to the framework management with an inter-service event.

```
[FrameworkManagementSBB] <> triggering SequenceEval_13804780792980 for execution <>
[FrameworkManagementSBB] fire InterServiceEvent >>
[ServiceControlMSBB] << InterServiceEvent received << SequenceEval_13804780792980
[ServiceControlMSBB] fire InterServiceEvent >> SequenceEval_13804780792980 >>
[Sequence] << InterServiceEvent received << sequence#SequenceEval_13804780792980#1
[Sequence] fire InterServiceEvent >> sequence#SequenceEval_13804780792980#1 >>
[Empty] << InterServiceEvent received empty#SequenceEval_13804780792980#3
[Empty] fire InterServiceEvent-> empty#SequenceEval_13804780792980#3 >>
[ServiceControlMSBB] << InterServiceEvent received << SequenceEval_13804780792980
[ServiceControlMSBB] fire InterServiceEvent >> SequenceEval_13804780792980 >>
[FrameworkManagementSBB] << InterServiceEvent received
[FrameworkManagementSBB]  -> SequenceEval_13804780792980-> executed
```

Figure 8.22: Sequence execution evaluation log

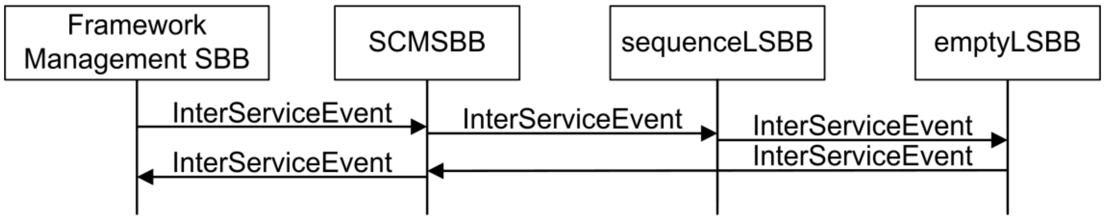


Figure 8.23: Sequence execution phase MSC

For both phases, the behaviour of the service instance is as expected. The Empty encapsulated within the Sequence activity was successfully parsed, and the correspondent LSBBs of the sequence and the empty activities created and executed.

8.3.3 Implementation of the Assign LSBB

The Assign LSBB supports initialisation, copy, and manipulation of variable values. It is possible to insert literals into the variable and to define expressions in order to insert and manipulate values.

An appropriate example BPEL process has been defined for the evaluation of the Assign LSBB. The graphical representation of this process is shown in Figure 8.24, and an excerpt of the corresponding XML representation in Figure 8.25. The name of the BPEL process is “AssignEval” (line 1). Two integer variables “var1” (line 13) and “var2” are defined for the assign of integer values to the variables (line 14). The process consists of two assign activities that are contained within the Sequence activity “main” (line 17 to 42).

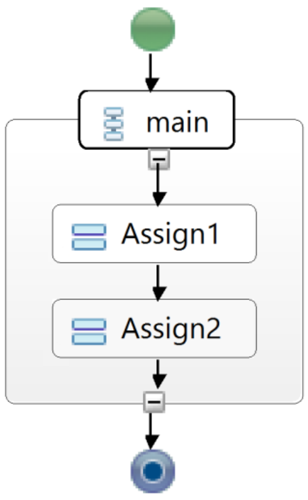
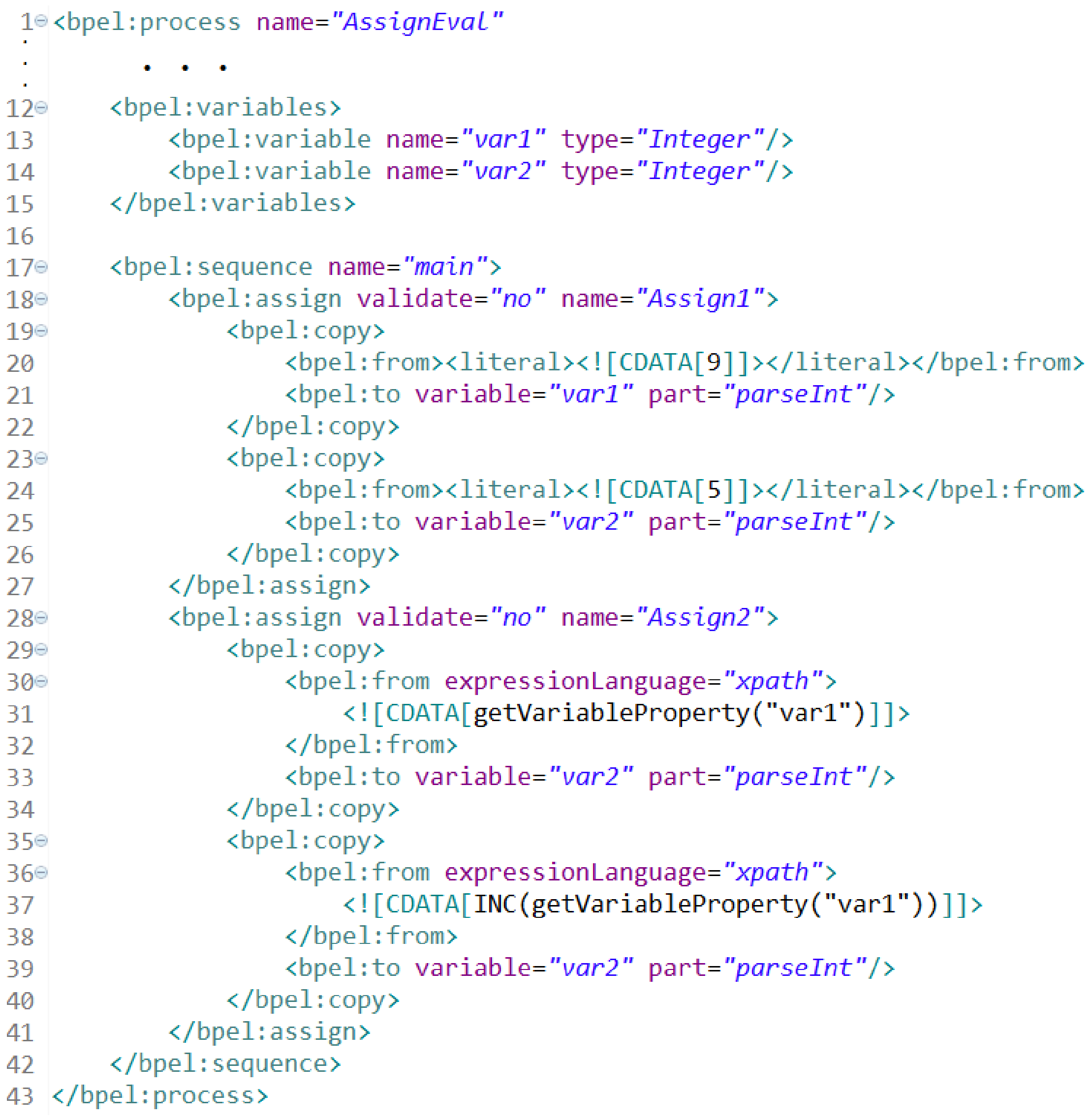


Figure 8.24: Graphical representation of the assign evaluation BPEL process

The first Assign activity “Assign1” (line 18 to 27) contains two “`copy`” operations. Both “`copy`” operations define literals in their “from” part that are copied into the variables described in the “to” part. Both variables are integer type variables. The literals are parsed with the “`parseInt`” operation, and their integer values are assigned to the variables. The first operation copies the literal “9” to the variable “var1” (line 19 to 22), and the second operation the literal “5” to variable “var2” (line 23 to 26).

The name of the second Assign activity is “Assign2” (line 28 to 41). It contains two “`copy`” operations. The first one (line 29 to 34) shows how a variable value can be read from a variable with the operation “`getVariableProperty(…)`”. Here the value of the variable “var1” is copied to the value of the variable “var2”. In the second “`copy`” operation (line 35 to 40), the value of variable “var1” is read with the “`getVariableProperty(…)`” operation and incremented with the “`INC(…)`” operation. The result is stored in variable “var2”.



```
<bpel:process name="AssignEval"
        . . .
    <bpel:variables>
        <bpel:variable name="var1" type="Integer"/>
        <bpel:variable name="var2" type="Integer"/>
    </bpel:variables>

    <bpel:sequence name="main">
        <bpel:assign validate="no" name="Assign1">
            <bpel:copy>
                <bpel:from><literal><![CDATA[9]]></literal></bpel:from>
                <bpel:to variable="var1" part="parseInt"/>
            </bpel:copy>
            <bpel:copy>
                <bpel:from><literal><![CDATA[5]]></literal></bpel:from>
                <bpel:to variable="var2" part="parseInt"/>
            </bpel:copy>
        </bpel:assign>
        <bpel:assign validate="no" name="Assign2">
            <bpel:copy>
                <bpel:from expressionLanguage="xpath">
                    <![CDATA[getVariableProperty("var1")]]>
                </bpel:from>
                <bpel:to variable="var2" part="parseInt"/>
            </bpel:copy>
            <bpel:copy>
                <bpel:from expressionLanguage="xpath">
                    <![CDATA[INC(getVariableProperty("var1"))]]>
                </bpel:from>
                <bpel:to variable="var2" part="parseInt"/>
            </bpel:copy>
        </bpel:assign>
    </bpel:sequence>
</bpel:process>
```

Figure 8.25: XML document excerpt of the “AssignEval” process

The service description is transferred to the service repository and triggered for creation and execution by the web interface of the Interactive Management Servlet. The log output of the service composition phase is given in Figure 8.26.

In the composition phase, it is expected that the SCMSBB parse the service description. All service components, which are found in the description, are created and configured.

```
[FrameworkManagementSBB] fire ConfigurationEvent >>
[ServiceControlMSBB] << ConfigurationEvent received
[ServiceControlMSBB] service name:AssignEval
[ServiceControlMSBB] create new service instance -> parsing description
[ServiceControlMSBB] found imports
[ServiceControlMSBB] variables:
[ServiceControlMSBB] Variable: Integer TypeClass: java.lang.Integer
[ServiceControlMSBB] Variable: Integer TypeClass: java.lang.Integer
[ServiceControlMSBB] found element -> sequence -> name: main
[ServiceControlMSBB] ElementIdentifier: sequence#AssignEval_21544234331296#6
[ServiceControlMSBB] fire ConfigurationEvent >> sequence#AssignEval_21544234331296#6 >>
[ServiceControlMSBB] found element -> assign -> name: Assign1
[ServiceControlMSBB] ElementIdentifier: assign#AssignEval_21544234331296#8
[ServiceControlMSBB] fire ConfigurationEvent >> assign#AssignEval_21544234331296#8 >>
[ServiceControlMSBB] found element -> assign -> name: Assign2
[ServiceControlMSBB] ElementIdentifier: assign#AssignEval_21544234331296#10
[ServiceControlMSBB] fire ConfigurationEvent >> assign#AssignEval_21544234331296#10 >>
[Sequence] << ConfigurationEvent received sequence#AssignEval_21544234331296#6
[Sequence] fire ReadyEvent >> sequence#AssignEval_21544234331296#6 >>
[ServiceControlMSBB] << ReadyEvent received! << sequence#AssignEval_21544234331296#6
[ServiceControlMSBB] Waiting for elements to become ready!
[Assign] << ConfigurationEvent received assign#AssignEval_21544234331296#8
[Assign] fireReadyEvent >> assign#AssignEval_21544234331296#8 >>
[ServiceControlMSBB] << ReadyEvent received! << assign#AssignEval_21544234331296#8
[ServiceControlMSBB] Waiting for elements to become ready!
[Assign] << ConfigurationEvent received assign#AssignEval_21544234331296#10
[Assign] fireReadyEvent >> assign#AssignEval_21544234331296#10 >>
[ServiceControlMSBB] << ReadyEvent received! << assign#AssignEval_21544234331296#10
[ServiceControlMSBB] fire ReadyEvent >> AssignEval_21544234331296
[FrameworkManagementSBB] << ReadyEvent received << AssignEval_21544234331296
[FrameworkManagementSBB] <---> AssignEval_21544234331296 created and configured <--->
```

Figure 8.26: Assign composition evaluation log

The framework management triggers the service composition by sending a configuration event to the SCMSBB. The SCMSBB parses the service description and identifies the required LSBBs. Then configuration events are sent to all three LSBBs, the Sequence LSBB and the two Assign LSBBs. Upon their creation and configuration, ready events are returned to the SCMSBB, and the SCMSBB returns a ready event to the framework management to confirm the creation and configuration of the service instance. The MSC with the event communication within the composition phase is shown in Figure 8.27.

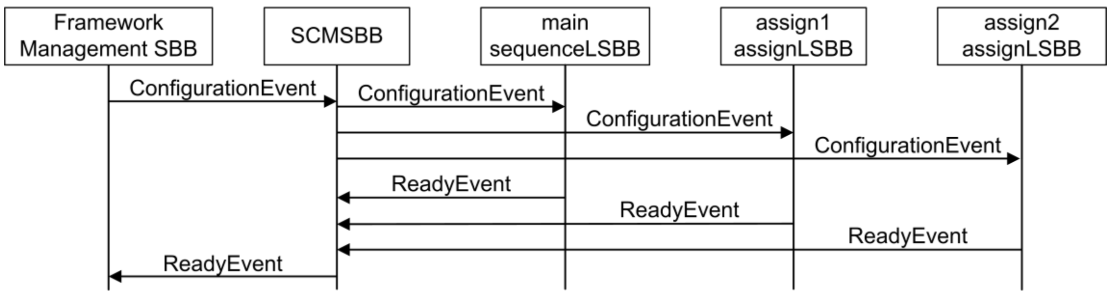


Figure 8.27: Assign composition phase MSC

It is expected that the components are executed during the execution phase in the same order as they are described in the process workflow. The Assign LSBBs should copy the variable values described in the "`copy`" operation "from" part to the variables described in the "`copy`" operation "to" part. For the assign evaluation, the results of the "`copy`" operations are printed into the log output (Figure 8.28).

```
[FrameworkManagementSBB] <> triggering AssignEval_21544234331296 for execution <>
[FrameworkManagementSBB] fire InterServiceEvent >>
[ServiceControlMSBB] << InterServiceEvent received << AssignEval_21544234331296
[ServiceControlMSBB] fire InterServiceEvent >> AssignEval_21544234331296 >>
[Sequence] << InterServiceEvent received << sequence#AssignEval_21544234331296#6
[Sequence] fire InterServiceEvent >> sequence#AssignEval_21544234331296#6 >>
[Assign] << InterServiceEvent received assign#AssignEval_21544234331296#8
[Assign]  Copy operation: new TO variable value: 9
[Assign]  Copy operation: new TO variable value: 5
[Assign] fire InterServiceEvent >> assign#AssignEval_21544234331296#8 >>
[Assign] << InterServiceEvent received assign#AssignEval_21544234331296#10
[Assign]  Copy operation: new TO variable value: 9
[Assign]  Copy operation: new TO variable value: 10
[Assign] fire InterServiceEvent >> assign#AssignEval_21544234331296#10 >>
[ServiceControlMSBB] << InterServiceEvent received << AssignEval_21544234331296
[ServiceControlMSBB] fire InterServiceEvent >> AssignEval_21544234331296 >>
[FrameworkManagementSBB] << InterServiceEvent received
[FrameworkManagementSBB]  -> AssignEval_21544234331296-> executed
```

Figure 8.28: Assign execution evaluation log

The service execution phase is triggered by an inter-service event from the framework management. The SCMSBB, which receives this inter-service event, activates the Sequence LSBB. The Sequence LSBB sends an inter-service event to the first Assign LSBB. The first "`copy`" operation copies the "9" into the variable "var1". As expected, the log output for this "`copy`" operation prints out the value

"9" as new value of variable "var1". In the next "`copy`" operation "5" is copied to the variable "var2" and the result "5" printed into the log output.

After finishing both "`copy`" operations of the first Assign LSBB, the second Assign LSBB is triggered. The first "`copy`" operation of "Assign2" reads the value from variable "var1" and copies it into variable "var2". The new value of variable "var2", which is also printed to the log output, is "9". The last "`copy`" operation again reads out the variable "var1". The variable is incremented by "1" and copied into variable "var2". The log output for the second "`copy`" operation is "10" (9 + 1). The MSC with the event communication of the execution phase is shown in Figure 8.29.

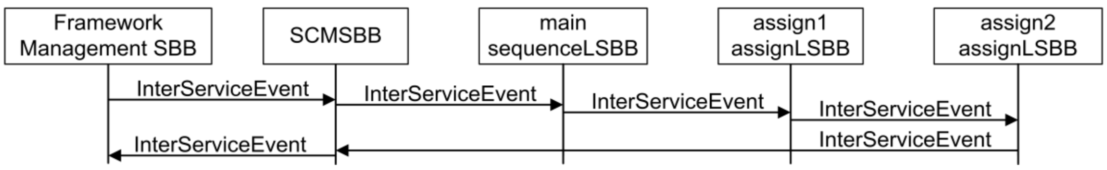


Figure 8.29: Assign execution phase MSC

Once the two "`copy`" operations of the last assign are executed, an inter-service event is sent to the SCMSBB. To inform the service management that the execution of the service instance has been completed, the SCMSBB sends an inter-service event to the framework management.

This section has verified the first basic functionality of the Assign LSBBs; further functionalities of the Assign LSBB will be evaluated in section 8.3.5, section 8.3.6, section 8.3.7, section 8.3.9, and in the Wake-up scenario in section 8.4.

8.3.4 Implementation of the Flow LSBB

The Flow activity provides the service developer with the possibility to describe parallel execution. This activity can consist of multiple branches that can be executed in parallel. Each branch can include further activities. The Flow activity is completed after all branches with their activities have been executed.

The BPEL Flow activity supports parallel execution as well. The Flow activity has been selected to demonstrate the parallel execution capabilities of service components in the prototype. The activities of a branch are encapsulated within a Sequence activity. With these sequences, also multiple activities within a flow branch are supported. Furthermore, these sequences can contain flow activities.

A minimal example BPEL process with one Flow activity has been chosen for the proof of concept. The graphical representation is shown in Figure 8.30, and the XML document representation can be seen in Figure 8.31.

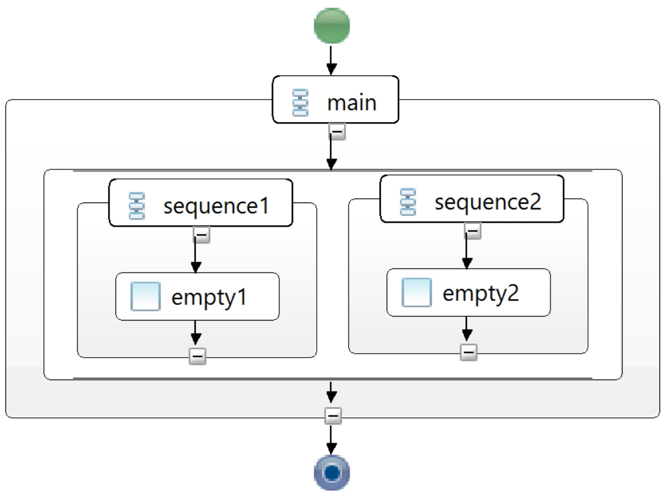
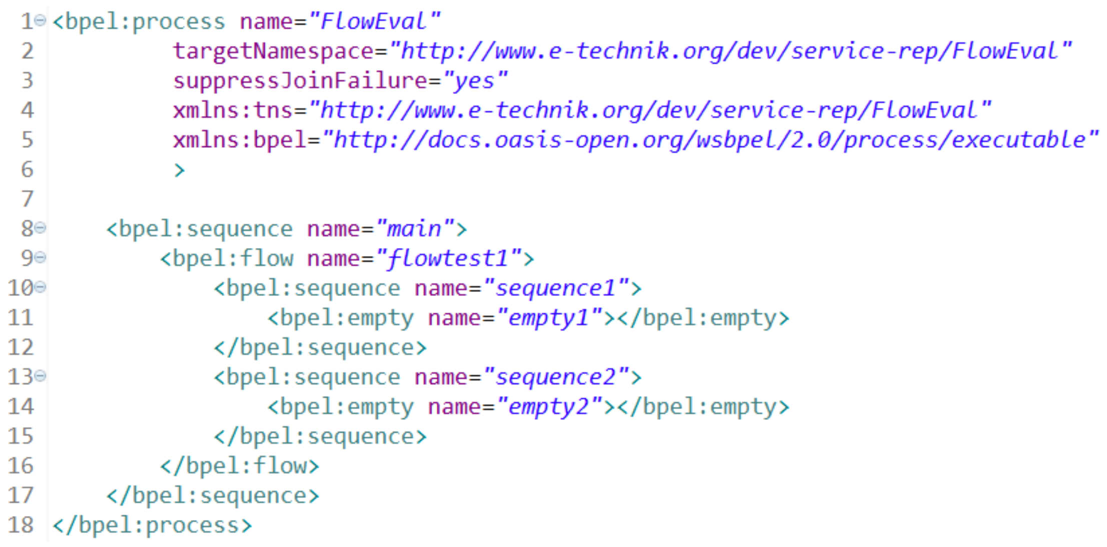


Figure 8.30: Graphical representation of the flow evaluation BPEL process

The name of the BPEL process is “FlowEval” (line 1). The first activity of the process is the Sequence activity “main” (line 8 to 17). It contains a Flow activity “flowtest1” (line 9 to 16), which consists of two branches. Each branch contains a Sequence activity (“sequence1” (line 10 to 12) and “sequence2” (line 13 to 15) with an Empty activity (“empty1” (line 10) and “empty2” (line 14)).



```
1  <bpel:process name="FlowEval"
2          targetNamespace="http://www.e-technik.org/dev/service-rep/FlowEval"
3          suppressJoinFailure="yes"
4          xmlns:tns="http://www.e-technik.org/dev/service-rep/FlowEval"
5          xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
6          >
7  
8      <bpel:sequence name="main">
9          <bpel:flow name="flowtest1">
10             <bpel:sequence name="sequence1">
11                 <bpel:empty name="empty1"></bpel:empty>
12             </bpel:sequence>
13             <bpel:sequence name="sequence2">
14                 <bpel:empty name="empty2"></bpel:empty>
15             </bpel:sequence>
16         </bpel:flow>
17     </bpel:sequence>
18 </bpel:process>
```

Figure 8.31: XML document excerpt of the “FlowEval” process

For the composition phase, it is expected that a service instance can be generated from the service description. In the execution phase, the two sequences within the flow together with their included components are executed in parallel. The log output of the composition phase is illustrated in Figure 8.32 and of the service execution phase in Figure 8.34. The event communication between the service components is shown in Figure 8.33 for the service composition phase and in Figure 8.35 for the service execution phase.

```
[FrameworkManagementSBB] fire ConfigurationEvent >>
[ServiceControlMSBB] << ConfigurationEvent received
[ServiceControlMSBB] service name:FlowEval
[ServiceControlMSBB] create new service instance -> parsing description
[ServiceControlMSBB] found element -> sequence -> name: main
[ServiceControlMSBB] ElementIdentifier: sequence#FlowEval_24640104818141#13
[ServiceControlMSBB] fire ConfigurationEvent >> sequence#FlowEval_24640104818141#13 >>
[ServiceControlMSBB] found element -> flow -> name: flowtest1
[ServiceControlMSBB] found element -> sequence -> name: seqflow1
[ServiceControlMSBB] ElementIdentifier: sequence#FlowEval_24640104818141#18
[ServiceControlMSBB] fire ConfigurationEvent >> sequence#FlowEval_24640104818141#18 >>
[ServiceControlMSBB] found element -> empty -> name: Empty1
[ServiceControlMSBB] ElementIdentifier: empty#FlowEval_24640104818141#20
[ServiceControlMSBB] fire ConfigurationEvent >> empty#FlowEval_24640104818141#20 >>
[ServiceControlMSBB] found element -> sequence -> name: seqflow2
[ServiceControlMSBB] ElementIdentifier: sequence#FlowEval_24640104818141#23
[ServiceControlMSBB] fire ConfigurationEvent >> sequence#FlowEval_24640104818141#23 >>
[ServiceControlMSBB] found element -> empty -> name: Empty2
[ServiceControlMSBB] ElementIdentifier: empty#FlowEval_24640104818141#25
[ServiceControlMSBB] fire ConfigurationEvent >> empty#FlowEval_24640104818141#25 >>
[ServiceControlMSBB] ElementIdentifier: flow#FlowEval_24640104818141#15
[ServiceControlMSBB] fire ConfigurationEvent >> flow#FlowEval_24640104818141#15 >>
[Sequence] << ConfigurationEvent received sequence#FlowEval_24640104818141#13
[Sequence] fire ReadyEvent >> sequence#FlowEval_24640104818141#13 >>
[ServiceControlMSBB] << ReadyEvent received! << sequence#FlowEval_24640104818141#13
[ServiceControlMSBB] Waiting for elements to become ready!
[Sequence] << ConfigurationEvent received sequence#FlowEval_24640104818141#18
[Sequence] fire ReadyEvent >> sequence#FlowEval_24640104818141#18 >>
[ServiceControlMSBB] << ReadyEvent received! << sequence#FlowEval_24640104818141#18
[ServiceControlMSBB] Waiting for elements to become ready!
[Empty] << ConfigurationEvent received empty#FlowEval_24640104818141#20
[Empty] fire ReadyEvent-> empty#FlowEval_24640104818141#20 >>
[ServiceControlMSBB] << ReadyEvent received! << empty#FlowEval_24640104818141#20
[ServiceControlMSBB] Waiting for elements to become ready!
[Sequence] << ConfigurationEvent received sequence#FlowEval_24640104818141#23
[Sequence] fire ReadyEvent >> sequence#FlowEval_24640104818141#23 >>
[ServiceControlMSBB] << ReadyEvent received! << sequence#FlowEval_24640104818141#23
[ServiceControlMSBB] Waiting for elements to become ready!
[Empty] << ConfigurationEvent received empty#FlowEval_24640104818141#25
[Empty] fire ReadyEvent-> empty#FlowEval_24640104818141#25 >>
[ServiceControlMSBB] << ReadyEvent received! << empty#FlowEval_24640104818141#25
[ServiceControlMSBB] Waiting for elements to become ready!
[Flow] << ConfigurationEvent received flow#FlowEval_24640104818141#15
[Flow] fire ReadyEvent >> flow#FlowEval_24640104818141#15 >>
[ServiceControlMSBB] << ReadyEvent received! << flow#FlowEval_24640104818141#15
[ServiceControlMSBB] fire ReadyEvent >> FlowEval_24640104818141
[FrameworkManagementSBB] << ReadyEvent received << FlowEval_24640104818141
[FrameworkManagementSBB] <---> FlowEval_24640104818141 created and configured <--->
```

Figure 8.32: Flow composition evaluation log

In the composition phase (Figure 8.33), the framework management triggers the SCMSBB, which analyses the service description. For each component found in the description, it sends configuration events to create and configure these components. The components confirm their creation and configuration with a ready event. Upon the SCMSBB has received ready events from all components, it will send a ready event to the framework management to confirm that the service instance is ready for execution.

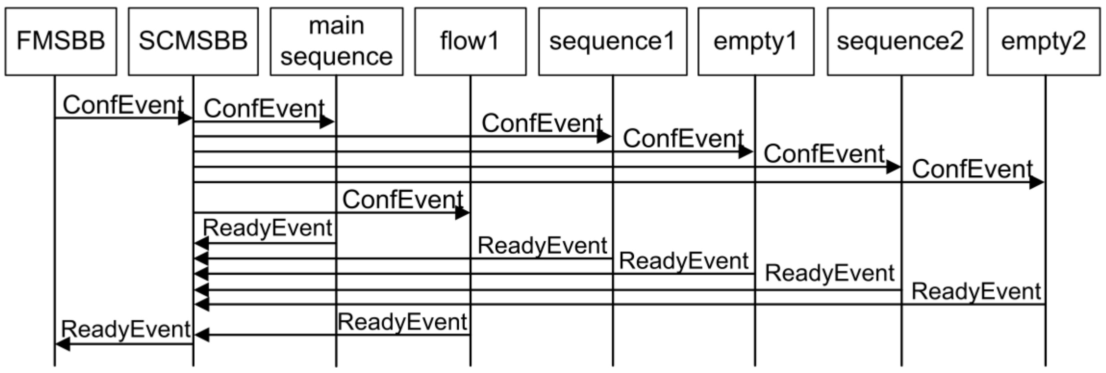


Figure 8.33: Flow composition phase MSC

The created service instance can now be executed (Figure 8.34 and Figure 8.35). To start the service, the framework management send an inter-service event to the SCMSBB. The SCMSBB triggers the first LSBB of the service instance, the Sequence LSBB, and the Sequence LSBB triggers the Flow LSBB.

For each branch of the Flow activity, the Flow LSBB sends an inter-service event to the contained Sequence LSBBs. The Sequence LSBBs start their execution independent from each other. Both Sequence LSBBs with their contained Empty LSBBs are executed in parallel.

```
[FrameworkManagementSBB] <> triggering FlowEval_24640104818141 for execution <>
[FrameworkManagementSBB] fire InterServiceEvent >>
[ServiceControlMSBB] << InterServiceEvent received << FlowEval_24640104818141
[ServiceControlMSBB] fire InterServiceEvent >> FlowEval_24640104818141 >>
[Sequence] << InterServiceEvent received << sequence#FlowEval_24640104818141#13
[Sequence] fire InterServiceEvent >> sequence#FlowEval_24640104818141#13 >>
[Flow] << InterServiceEvent received << flow#FlowEval_24640104818141#15
[Flow] fire InterServiceEvent >> flow#FlowEval_24640104818141#15 >>
[Flow] fire InterServiceEvent >> flow#FlowEval_24640104818141#15 >>
[Sequence] << InterServiceEvent received << sequence#FlowEval_24640104818141#18
[Sequence] fire InterServiceEvent >> sequence#FlowEval_24640104818141#18 >>
[Empty] << InterServiceEvent received empty#FlowEval_24640104818141#20
[Sequence] << InterServiceEvent received << sequence#FlowEval_24640104818141#23
[Empty] fire InterServiceEvent-> empty#FlowEval_24640104818141#20 >>
[Sequence] fire InterServiceEvent >> sequence#FlowEval_24640104818141#23 >>
[Empty] << InterServiceEvent received empty#FlowEval_24640104818141#25
[Empty] fire InterServiceEvent-> empty#FlowEval_24640104818141#25 >>
[Flow] << InterServiceEvent received << flow#FlowEval_24640104818141#15
[Flow] << InterServiceEvent received << flow#FlowEval_24640104818141#15
[Flow] fire InterServiceEvent >> flow#FlowEval_24640104818141#15 >>
[ServiceControlMSBB] << InterServiceEvent received << FlowEval_24640104818141
[ServiceControlMSBB] fire InterServiceEvent >> FlowEval_24640104818141 >>
[FrameworkManagementSBB] << InterServiceEvent received
[FrameworkManagementSBB]  -> FlowEval_24640104818141-> executed
```

Figure 8.34: Flow execution evaluation log

The Flow LSBB waits until all branches have completed their execution before it sends out an inter-service event to the SCMSBB. The SCMSBB confirms the execution of the service instance back to the framework management with an inter-service event.

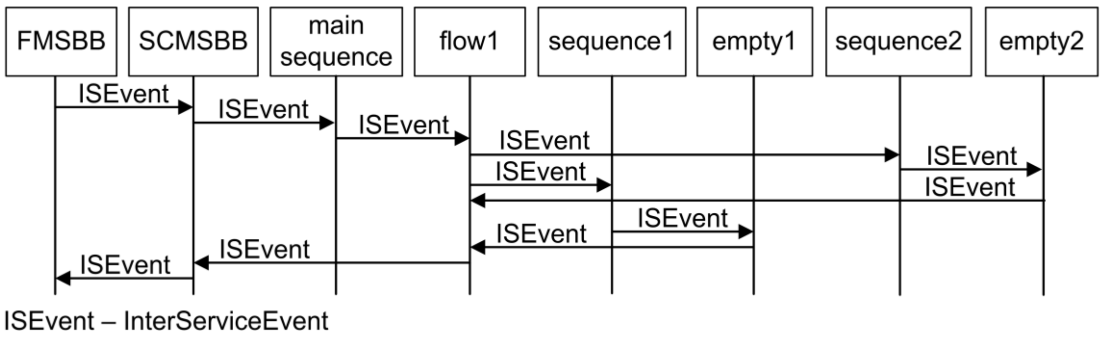


Figure 8.35: Flow execution phase MSC

The example BPEL process demonstrates the possibility of the framework for parallel execution of service components within the same service instance.

8.3.5 Implementation of the IF LSBB

Conditional behaviour can be described with the If activity. It contains a list of one or more conditional branches defined by the "if" condition and the optional "elseif" and "else" conditions. The order in the list of branches is also the order in which the conditions are analysed. If a condition is evaluated to be true, the corresponding branch is executed. If a condition evaluates to be false, the next condition is analysed. If no condition evaluates to be true, the "else" branch is executed. The If LSBB is completed, when the components contained in the selected branch have been executed, or immediately when no condition evaluates to be true and no "else" branch is specified.

The If LSBB is evaluated with the help of a BPEL process, the graphical representation of the process is depicted in Figure 8.36, and the XML document of this process is shown in Figure 8.37.

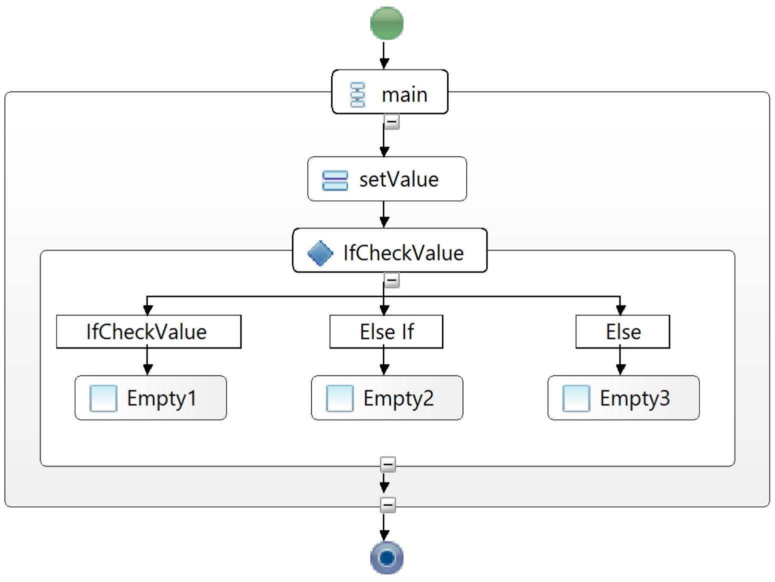
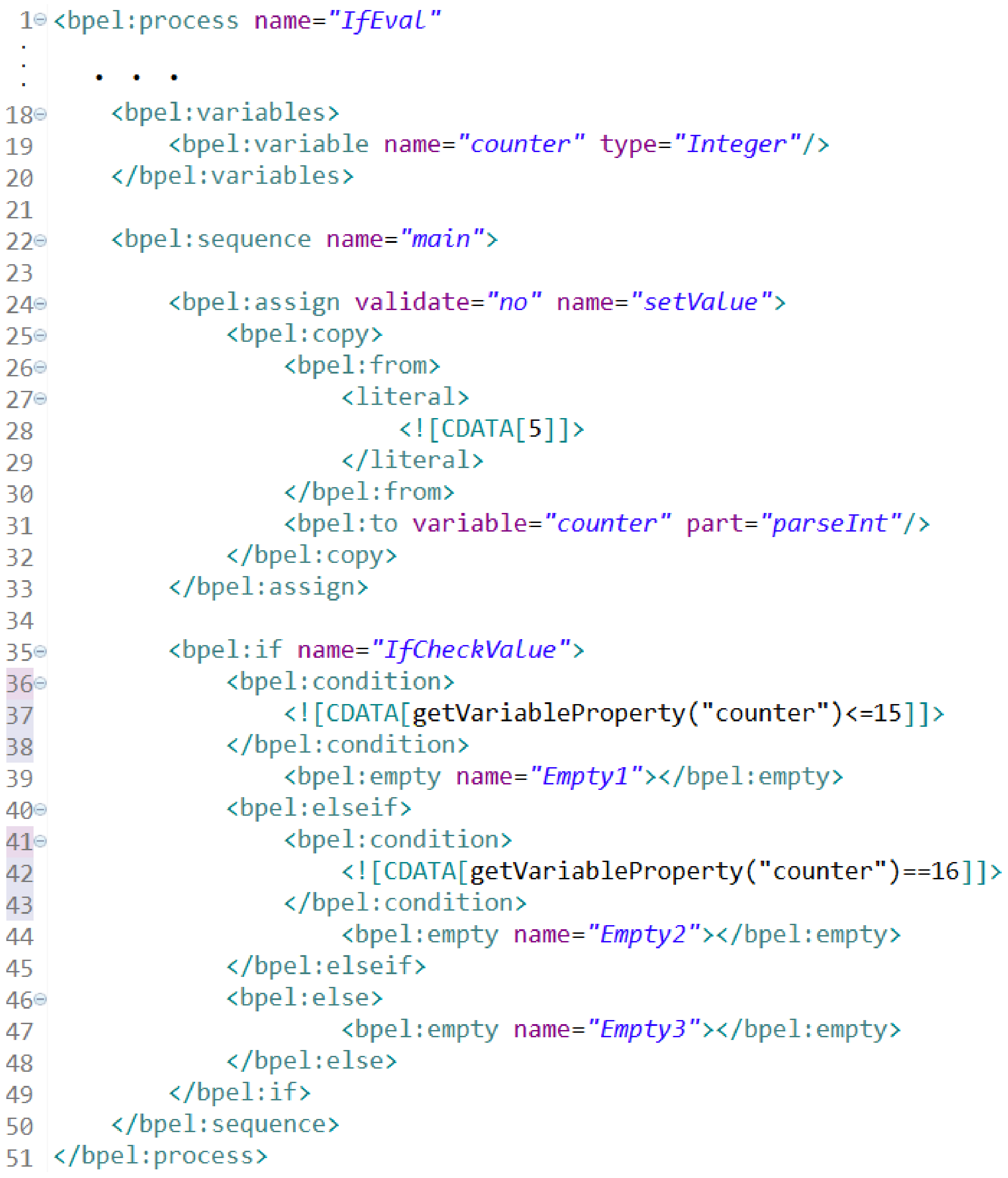


Figure 8.36: Graphical representation of the if evaluation BPEL process

The process “IfEval” (line 1) contains six activities, a Sequence activity “main” (line 22 to 50), an Assign activity “setValue” (line 24 to 33), the If activity “IfCheckValue” (line 35 to 49), and three empty activities, namely “Empty1” (line 39), “Empty2” (line 44), and “Empty3” (line 47). The Sequence activity contains the other five activities. An integer variable with the name “counter” (line 19) is defined within the process. In the Assign activity, this variable is initialised with the value “5”. The If activity contains three branches, one “if” branch (line 35 to 39), one “elseif” branch (line 40 to 45), and one “else” branch (line 46 to 48). As described in section 8.2, the prototype supports some example operations, expressions, and conditions. The condition in the “if” branch reads out the value of the variable “counter”. To do this, it uses the “`getVariableProperty(…)`” operation.



```
<bpel:process name="IfEval"

    •  •  •
    <bpel:variables>
        <bpel:variable name="counter" type="Integer"/>
    </bpel:variables>

    <bpel:sequence name="main">

        <bpel:assign validate="no" name="setValue">
            <bpel:copy>
                <bpel:from>
                    <literal>
                        <![CDATA[5]]>
                    </literal>
                </bpel:from>
                <bpel:to variable="counter" part="parseInt"/>
            </bpel:copy>
        </bpel:assign>

        <bpel:if name="IfCheckValue">
            <bpel:condition>
                <![CDATA[getVariableProperty("counter")<=15]]>
            </bpel:condition>
                <bpel:empty name="Empty1"></bpel:empty>
            <bpel:elseif>
                <bpel:condition>
                    <![CDATA[getVariableProperty("counter")==16]]>
                </bpel:condition>
                    <bpel:empty name="Empty2"></bpel:empty>
            </bpel:elseif>
            <bpel:else>
                    <bpel:empty name="Empty3"></bpel:empty>
            </bpel:else>
        </bpel:if>
    </bpel:sequence>
</bpel:process>
```

Figure 8.37: XML document excerpt of the "IfEval" process

In the "if" condition, the "counter" variable is analysed. If the value is lower than or equal (<=) to "15", the condition is evaluated to true and the "if" branch with the "Empty1" activity is executed. If the condition is evaluated to false, the "elseif" condition is analysed. In this condition, the variable value is compared with "16". If the value is equal (==) to "16", the condition evaluates to true and the "Empty2" activity contained in the “elseif” branch is executed. In case that the “elseif” branch evaluates to false, the “Empty3” activity in the “else” branch is executed.

In the composition phase, it is expected that all LSBBs are created and configured, including all three “if” branches. In the execution phase it is expected that only the first “if” branch is executed. The “if” branch evaluates to true because the value of the “counter” variable is set to “5”, and the “if” condition evaluates to “true” when the variable value is less than or equal to “15”. After the “if” branch with the “Empty1” LSBB has been executed, an inter-service event should be sent back to the SCMSBB. The other branches should not be executed.

For the evaluation of the IF LSBB, the service description is loaded into the service repository, and the service is triggered for configuration and execution by the web interface of the Interactive Management Servlet. The log outputs of the service composition phase and of the service execution phase are analysed. The log output of the composition phase is given in Figure 8.38 and Figure 8.39, the corresponding MSC showing the event communication in Figure 8.40. The log output of the service execution phase is shown in Figure 8.41 and the MSC with the event communication in Figure 8.42.

The framework management triggers the composition phase with a configuration event (see Figure 8.38 and Figure 8.40). The SCMSBB analyses the service description and identifies the required components and variables. For creation and configuration of the LSBBs, configuration events are sent to these components.

```
[FrameworkManagementSBB] fire ConfigurationEvent >>
[ServiceControlMSBB] << ConfigurationEvent received
[ServiceControlMSBB] service name:IfEval
[ServiceControlMSBB] create new service instance -> parsing description
[ServiceControlMSBB] found imports
[ServiceControlMSBB] found imports
[ServiceControlMSBB] partnerLinks:
[ServiceControlMSBB] variables:
[ServiceControlMSBB] Variable: Integer TypeClass: java.lang.Integer
[ServiceControlMSBB] found element -> sequence -> name: main
[ServiceControlMSBB] ElementIdentifier: sequence#IfEval_4749589343457#1
[ServiceControlMSBB] fire ConfigurationEvent >> sequence#IfEval_4749589343457#1 >>
[ServiceControlMSBB] found element -> assign -> name: setValue
[ServiceControlMSBB] ElementIdentifier: assign#IfEval_4749589343457#3
[ServiceControlMSBB] fire ConfigurationEvent >> assign#IfEval_4749589343457#3 >>
[ServiceControlMSBB] found element -> if -> name: IfCheckValue
[ServiceControlMSBB] elementIdentifier: if#IfEval_4749589343457#5
[ServiceControlMSBB] found element -> empty -> name: Empty1
[ServiceControlMSBB] ElementIdentifier: empty#IfEval_4749589343457#7
[ServiceControlMSBB] fire ConfigurationEvent >> empty#IfEval_4749589343457#7 >>
[ServiceControlMSBB] found element -> empty -> name: Empty2
[ServiceControlMSBB] ElementIdentifier: empty#IfEval_4749589343457#10
[ServiceControlMSBB] fire ConfigurationEvent >> empty#IfEval_4749589343457#10 >>
[ServiceControlMSBB] found element -> empty -> name: Empty3
[ServiceControlMSBB] ElementIdentifier: empty#IfEval_4749589343457#13
[ServiceControlMSBB] fire ConfigurationEvent >> empty#IfEval_4749589343457#13 >>
[ServiceControlMSBB] fire ConfigurationEvent >> if#IfEval_4749589343457#5 >>
```

Figure 8.38: If composition evaluations log – part 1

Once all service components have been created, they receive their configuration events (Figure 8.39 and Figure 8.40).

```
[Sequence] << ConfigurationEvent received sequence#IfEval_4749589343457#1
[Sequence] fire ReadyEvent >> sequence#IfEval_4749589343457#1 >>
[ServiceControlMSBB] << ReadyEvent received! << sequence#IfEval_4749589343457#1
[ServiceControlMSBB] Waiting for elements to become ready!
[Assign] << ConfigurationEvent received assign#IfEval_4749589343457#3
[Assign] fireReadyEvent >> assign#IfEval_4749589343457#3 >>
[ServiceControlMSBB] << ReadyEvent received! << assign#IfEval_4749589343457#3
[ServiceControlMSBB] Waiting for elements to become ready!
[Empty] << ConfigurationEvent received empty#IfEval_4749589343457#7
[Empty] fire ReadyEvent-> empty#IfEval_4749589343457#7 >>
[ServiceControlMSBB] << ReadyEvent received! << empty#IfEval_4749589343457#7
[ServiceControlMSBB] Waiting for elements to become ready!
[Empty] << ConfigurationEvent received empty#IfEval_4749589343457#10
[Empty] fire ReadyEvent-> empty#IfEval_4749589343457#10 >>
[ServiceControlMSBB] << ReadyEvent received! << empty#IfEval_4749589343457#10
[ServiceControlMSBB] Waiting for elements to become ready!
[Empty] << ConfigurationEvent received empty#IfEval_4749589343457#13
[Empty] fire ReadyEvent-> empty#IfEval_4749589343457#13 >>
[ServiceControlMSBB] << ReadyEvent received! << empty#IfEval_4749589343457#13
[ServiceControlMSBB] Waiting for elements to become ready!
[If] << ConfigurationEvent received << if#IfEval_4749589343457#5
[If] fire ReadyEvent >> if#IfEval_4749589343457#5
[ServiceControlMSBB] << ReadyEvent received! << if#IfEval_4749589343457#5
[ServiceControlMSBB] fire ReadyEvent >> IfEval_4749589343457
[FrameworkManagementSBB] << ReadyEvent received << IfEval_4749589343457
```

Figure 8.39: If composition evaluations log part – 2

Upon configuration, the service components return a ready event back to the SCMSBB. The SCMSBB sends a ready event back to the framework management when it has received the ready events of all other components.

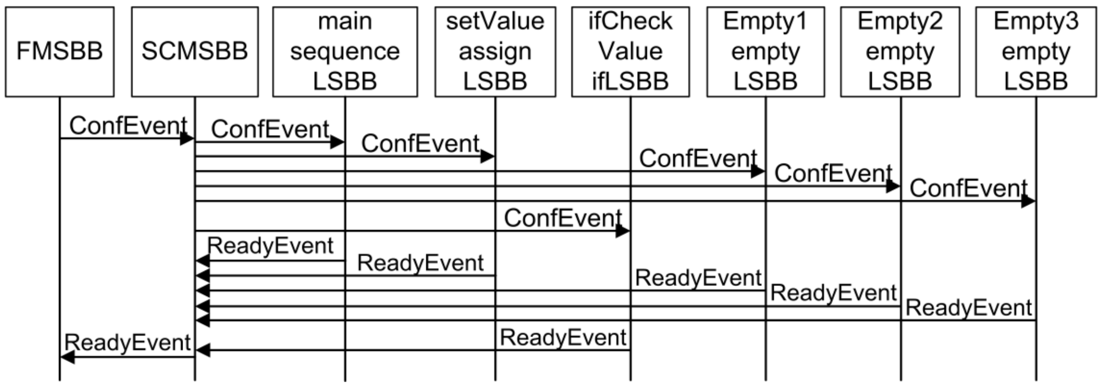


Figure 8.40: If composition phase MSC

The execution phase starts with an inter-service event from the framework management (Figure 8.41). The workflow of the service instance is executed in accordance to the service description, and inter-service events are sent from the executed LSBB to the subsequent one. The “counter” variable is initialised to “5” in the Assign LSBB and analysed in the “if” condition of the If LSBB. If the value of the variable is less than or equal to 15, the condition evaluates to be true. In this case the condition evaluates to “true”, and the “if” branch with the Empty1 activity is executed.

The execution of the “if” branch is finished after the execution of the Empty1 LSBB. This LSBB sends back an inter-service event to the SCMSBB. The other empty LSBBs, Empty2 LSBB and Empty3 LSBB, are not executed, because the “if” condition already evaluates to true. The SCMSBB sends an inter-service event to the framework management, and the execution of the service instance is completed.

```
[FrameworkManagementSBB] <> triggering IfEval_4749589343457 for execution <>
[FrameworkManagementSBB] fire InterServiceEvent >>
[ServiceControlMSBB] << InterServiceEvent received << IfEval_4749589343457
[ServiceControlMSBB] fire InterServiceEvent >> IfEval_4749589343457 >>
[Sequence] << InterServiceEvent received << sequence#IfEval_4749589343457#1
[Sequence] fire InterServiceEvent >> sequence#IfEval_4749589343457#1 >>
[Assign] << InterServiceEvent received assign#IfEval_4749589343457#3
[Assign]  Copy operation: new TO variable value: 15
[Assign] fire InterServiceEvent >> assign#IfEval_4749589343457#3 >>
[If] << InterServiceEvent received << if#IfEval_4749589343457#5
[If] Condition (if): getVariableProperty("counter")<=15 is true
[If] fire InterServiceEvent >> if#IfEval_4749589343457#5
[Empty] << InterServiceEvent received empty#IfEval_4749589343457#7
[Empty] fire InterServiceEvent-> empty#IfEval_4749589343457#7 >>
[ServiceControlMSBB] << InterServiceEvent received << IfEval_4749589343457
[ServiceControlMSBB] fire InterServiceEvent >> IfEval_4749589343457 >>
[FrameworkManagementSBB] << InterServiceEvent received
[FrameworkManagementSBB]  -> IfEval_4749589343457-> executed
```

Figure 8.41: If execution evaluation log

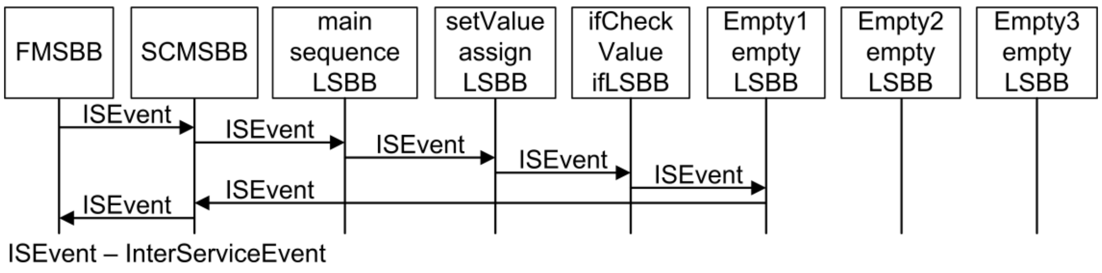


Figure 8.42: If execution phase MSC

8.3.6 Implementation of the Invoke LSBB together with SIP RCSBB

In BPEL, the Invoke LSBB is represented by an Invoke activity. It is used to call methods within a partner link. In the SEE, it communicates with a RCSBB. This RCSBB implements the methods that offer the service functionalities and protocol specific communication though the RAs.

An example BPEL process is used for the proof of concept of the Invoke LSBB. The graphical representation of the BPEL process is shown in Figure 8.43. Excerpts of the corresponding XML representation of the process are given in Figure 8.44, Figure 8.45, Figure 8.46, and Figure 8.47.

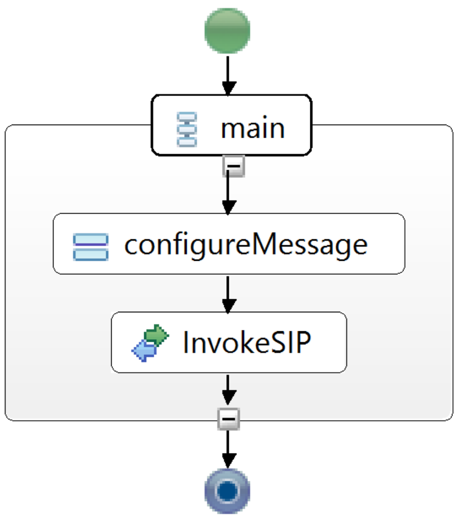


Figure 8.43: Graphical representation of the invoke evaluation BPEL process

The BPEL process describes the configuration and sending of a SIP MESSAGE request. Only three BPEL activities are required for this process; a Sequence activity that contains the other two activities, one Assign activity (line 28 to59, Figure 8.46) for a creation of the SIP MESSAGE request, and one Invoke activity (line 60 to 65, Figure 8.47) for sending the SIP request.

For this service, the SIP protocol will be used, and a CBB that supports this protocol is required. Therefore, for the prototype a CBB called “SIP CBB” has been developed that supports the functionalities required for this service.

The SIP CBB consists of the SIP RCSBB, the (JAIN SLEE) SIP RA, the “SIPServices” partner link, the SIP request, and SIP response data types. The SIP CBB partner link description is shown in Figure 8.44.

```
<bpel:partnerLinks>
    <bpel:partnerLink name="SIPServices"
        partnerLinkType="tns:InvokeEval"
        myRole="InvokeEvalProvider" />
</bpel:partnerLinks>
```

Figure 8.44: SIP CBB partner link from the "InvokeEval" process – part 1

Before the SIP MESSAGE request can be sent, it must be configured. Therefore, a "sipRequestType" variable is required. In Figure 8.45, this variable is created with the name "sendSIPRequest". The complex "sipRequestType" variable belongs to the SIPCBB and consists of multiple parts. This allows the configuration of the SIP request header fields. The implemented SIP CBB will set the most header fields automatically, but some header fields can be manipulated in BPEL by the service developer, e.g., the "From" and "To" display names, the "From" and "To" SIP URIs, the Contact SIP URI, and the SIP message body.

```
<bpel:variables>
    <bpel:variable name="sendSIPRequest" messageType="sipmt:sipRequestType"/>
</bpel:variables>
```

Figure 8.45: XML document excerpt of the "InvokeEval" process – part 2

The SIP request variable is configured within the Assign activity (Figure 8.46). The Assign activity consists of five "`copy`" operations. In the first operation (line 29 to 34), the "From" display name of the request is set to "testservice". The second "`copy`" operation (line 35 to 40) configures the "From" URI, the third "`copy`" operation (line 41 to 46) the "To" display name, and the fourth (line 47 to 52) the "To" URI of the request. In the last "`copy`" operation (line 53 to 58), the message body of the SIP MESSAGE request is set to "test message".

```
<bpel:assign validate="no" name="configureMessage">
        <bpel:copy>
            <bpel:from>
                <literal><![CDATA[testservice]]></literal>
            </bpel:from>
            <bpel:to variable="sendSIPRequest" part="SIPRequestFromDisplayName"/>
        </bpel:copy>
        <bpel:copy>
            <bpel:from>
                <literal><![CDATA[sip:testservice@192.168.67.49]]></literal>
            </bpel:from>
            <bpel:to variable="sendSIPRequest" part="SIPRequestFromURI"/>
        </bpel:copy>
        <bpel:copy>
            <bpel:from>
                <literal><![CDATA[testuser]]></literal>
            </bpel:from>
            <bpel:to variable="sendSIPRequest" part="SIPRequestToDisplayName"/>
        </bpel:copy>
        <bpel:copy>
            <bpel:from>
                <literal><![CDATA[sip:testuser@192.168.67.15]]></literal>
            </bpel:from>
            <bpel:to variable="sendSIPRequest" part="SIPRequestToURI"/>
        </bpel:copy>
        <bpel:copy>
            <bpel:from>
                <literal><![CDATA[test message]]></literal>
            </bpel:from>
            <bpel:to variable="sendSIPRequest" part="SIPMessageBody"/>
        </bpel:copy>
    </bpel:assign>
```

Figure 8.46: XML document excerpt of the "InvokeEval" process – part 3

The last activity of the BPEL process is the Invoke activity "InvokeSIP" (Figure 8.47). This activity invokes the operation "`doSIPMessage`" on the "SIPServices" partner link with the variable "sendSIPRequest". The Invoke activity describes the sending of the SIP Message request defined in the variable "sendSIPRequest".

```
<bpel:invoke name="InvokeSIP"
      operation="doSIPMessage"
      partnerLink="SIPServices"
      portType="tns:InvokeEval"
      variable=" sendSIPRequest "
      />
```

Figure 8.47: XML document excerpt of the "InvokeEval" process – part 4

The described service is loaded into the service repository of the SEE and is triggered by the framework management. In the composition phase, it is expected that all components of the service instance are created and configured, the SCMSBB, the three LSBBs (Sequence LSBB, Assign LSBB, and Invoke LSBB), and the SIP RCSBB, which is part of the SIP CBB and implements the required SIP functionally. In the execution phase, it is expected that the framework management triggers the service execution. The components of the service instance communicate with each other using inter-service events.

The Invoke LSBB triggers the SIP RCSBB to send out the configured SIP MESSAGE request. This RCSBB prepares the SIP Request, which is sent by the SIP RA of the SIP CBB. To fulfil the SIP transaction, a 200 OK response from the called SIP user agent is expected. The SIP RA handles the SIP 200 OK message. After the SIP message has been sent, the SIP RCSBB should fires an inter-service event to the SCMSBB, which upon reception also sends an inter-service event to the management framework to confirm the execution of the service instance.

The log output of the service composition phase is shown in Figure 8.48 and Figure 8.49. The corresponding MSC shows the event communication in Figure 8.50. For the service execution phase, the log output is given in Figure 8.51. The SIP MESSAGE request is illustrated in Figure 8.52 and the corresponding 200 OK response in Figure 8.53. The MSC with the event communication and the SIP transaction is shown in Figure 8.54.

```
[FrameworkManagementSBB] fire ConfigurationEvent >>
[ServiceControlMSBB] << ConfigurationEvent received -> #ID: AssignEval_13254597070766
[ServiceControlMSBB] service name:InvokeEval
[ServiceControlMSBB] create new service instance -> parsing description
[ServiceControlMSBB] found imports
[ServiceControlMSBB] found imports
[ServiceControlMSBB] partnerLinks:
[ServiceControlMSBB] partnerLink: -> name: SIPServices
[ServiceControlMSBB] variables:
[ServiceControlMSBB] Variable: sipmt:sipRequestType TypeClass: de.rcsbb.sip.SipRequestType
[ServiceControlMSBB] found element -> sequence -> name: main
[ServiceControlMSBB] ElementIdentifier: sequence#InvokeEval_13290178330801#8
[ServiceControlMSBB] fire ConfigurationEvent >> sequence#InvokeEval_13290178330801#8 >>
[ServiceControlMSBB] found element -> assign -> name: configureMessage
[ServiceControlMSBB] ElementIdentifier: assign#InvokeEval_13290178330801#10
[ServiceControlMSBB] fire ConfigurationEvent >> assign#InvokeEval_13290178330801#10 >>
[ServiceControlMSBB] found element -> invoke -> name: InvokeSIP
[ServiceControlMSBB] ElementIdentifier: invoke#InvokeEval_13290178330801#12
[ServiceControlMSBB] fire ConfigurationEvent >> invoke#InvokeEval_13290178330801#12 >>
[ServiceControlMSBB] found element -> rcsbb -> name: SIPServices
[ServiceControlMSBB] ElementIdentifier: SIPServices#InvokeEval_13290178330801#14
[ServiceControlMSBB] fire ConfigurationEvent >> SIPServices#InvokeEval_13290178330801#14 >>
```

Figure 8.48: Invoke composition evaluation log – part 1

As expected, the management framework triggers the service composition phase with a configuration event sent to the SCMSBB (Figure 8.48 and Figure 8.50). The SCMSBB analyses the service description and identifies the required variables and the components of the service instance. Configuration events are sent to create and configure the service components.

```
[Sequence] << ConfigurationEvent received sequence#InvokeEval_13290178330801#8
[Sequence] fire ReadyEvent >> sequence#InvokeEval_13290178330801#8 >>
[ServiceControlMSBB] << ReadyEvent received! << sequence#InvokeEval_13290178330801#8
[ServiceControlMSBB] Waiting for elements to become ready!
[Assign] << ConfigurationEvent received assign#InvokeEval_13290178330801#10
[Assign] fireReadyEvent >> assign#InvokeEval_13290178330801#10 >>
[ServiceControlMSBB] << ReadyEvent received! << assign#InvokeEval_13290178330801#10
[ServiceControlMSBB] Waiting for elements to become ready!
[Invoke] << ConfigurationEvent received << invoke#InvokeEval_13290178330801#12
[Invoke] fire ReadyEvent >> invoke#InvokeEval_13290178330801#12
[ServiceControlMSBB] << ReadyEvent received! << invoke#InvokeEval_13290178330801#12
[ServiceControlMSBB] Waiting for elements to become ready!
[SIP_RCSBB] << ConfigurationEvent received << SIPServices#InvokeEval_13290178330801#14
[SIP_RCSBB] fire ReadyEvent >> SIPServices#InvokeEval_13290178330801#14
[ServiceControlMSBB] << ReadyEvent received! << SIPServices#InvokeEval_13290178330801#14
[ServiceControlMSBB] fire ReadyEvent >> InvokeEval_13290178330801
[FrameworkManagementSBB] << ReadyEvent received << InvokeEval_13290178330801
[FrameworkManagementSBB] <---> InvokeEval_13290178330801 created and configured <--->
```

Figure 8.49: Invoke composition evaluation log – part 2

Each service component receives its configuration event, sets the new configuration, and returns a ready event to the SCMSBB. To confirm the creation and configuration of the SCMSBB, a ready event is also sent to the framework management (Figure 8.49, Figure 8.50). With this step, the service instance is ready for execution.

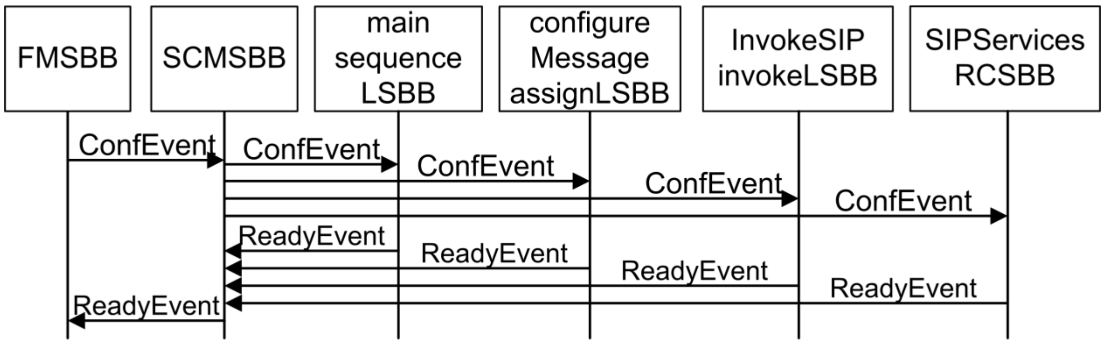


Figure 8.50: Invoke composition phase MSC

The execution phase is initiated with an inter-service event from the framework management to the SCMSBB (Figure 8.51, Figure 8.54). The SCMSBB activates the Sequence LSBB, which starts the Assign LSBB. The Assign LSBB configures the parameters of the request message variables. When finalised it sends an inter-service event to the Invoke LSBB that triggers the SIP RCSBB.

```
[FrameworkManagementSBB] <> triggering InvokeEval_13290178330801 for execution <>
[FrameworkManagementSBB] fire InterServiceEvent >>
[ServiceControlMSBB] << InterServiceEvent received << InvokeEval_13290178330801
[ServiceControlMSBB] fire InterServiceEvent >> InvokeEval_13290178330801 >>
[Sequence] << InterServiceEvent received << sequence#InvokeEval_13290178330801#8
[Sequence] fire InterServiceEvent >> sequence#InvokeEval_13290178330801#8 >>
[Assign] << InterServiceEvent received assign#InvokeEval_13290178330801#10
[Assign] fire InterServiceEvent >> assign#InvokeEval_13290178330801#10 >>
[Invoke] << InterServiceEvent received << invoke#InvokeEval_13290178330801#12
[Invoke] fire InterServiceEvent >> invoke#InvokeEval_13290178330801#12
[SIP_RCSBB] << InterServiceEvent received SIPServices#InvokeEval_13290178330801#14
[SIP_RCSBB] fire MessageEvent >> TO: sip:testuser@192.168.67.15
[SIP_RCSBB] fire InterServiceEvent >> SIPServices#InvokeEval_13290178330801#14
[ServiceControlMSBB] << InterServiceEvent received << InvokeEval_13290178330801
[ServiceControlMSBB] fire InterServiceEvent >> InvokeEval_13290178330801 >>
[FrameworkManagementSBB] << InterServiceEvent received
[FrameworkManagementSBB]  -> InvokeEval_13290178330801-> executed
```

Figure 8.51: Invoke execution evaluation log

The SIP RCSBB creates and configures the SIP MESSAGE request (Figure 8.52) and sends it out with the SIP RA. The SIP softphone that receives the SIP request (Figure 8.55) returns a 200 OK response (Figure 8.53) to the SIP RA. In the next step, the SIP RCSBB sends an inter-service event to the SCMSBB, which also returns an inter-service event to the framework management.

```
MESSAGE sip:testuser@192.168.67.15 SIP/2.0
Call-ID: cd11863c478ac0923afb9776757da7c3@192.168.67.49
CSeq: 1 MESSAGE
From: "testservice" <sip:testservice@192.168.67.49>;tag=98421389536618493192.168.67.49
To: "testuser" <sip:testuser@192.168.67.15>
Via: SIP/2.0/UDP 192.168.67.49:5060;branch=z9hG4bK-323037-8aa28fa5454ca00299e48b9016875983
Max-Forwards: 70
Content-Type: text/plain
Content-Length: 12

test message
```

Figure 8.52: Invoke evaluation log SIP MESSAGE

```
15:23:39,039 INFO  [SipResourceAdaptor] Received Response:
SIP/2.0 200 OK
Via: SIP/2.0/UDP 192.168.67.49:5060;branch=z9hG4bK-323037-8aa28fa5454ca00299e48b9016875983
From: "testservice" <sip:testservice@192.168.67.49>;tag=98421389536618493192.168.67.49
To: "testuser" <sip:testuser@192.168.67.15>;tag=802775cc027ae311a623005056c00008
Call-ID: cd11863c478ac0923afb9776757da7c3@192.168.67.49
CSeq: 1 MESSAGE
Contact: <sip:testuser@192.168.67.15:5060>
Allow: INVITE,OPTIONS,ACK,BYE,CANCEL,INFO,NOTIFY,MESSAGE,UPDATE
Server: SIPPER for PhonerLite
Date: Sun, 12 Jan 2014 14:23:39 GMT
Content-Length: 0
```

Figure 8.53: Invoke evaluation log SIP 200OK

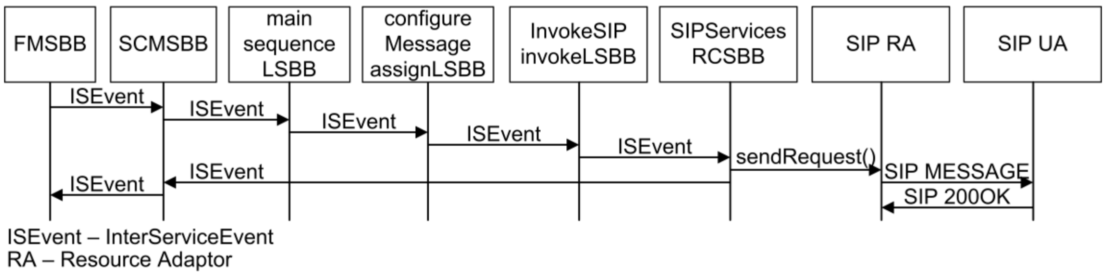
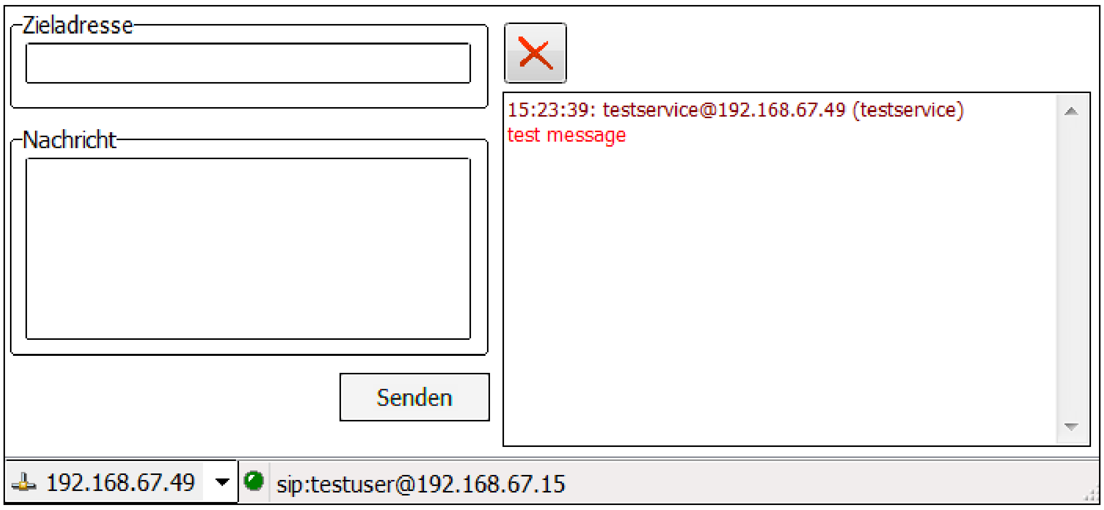


Figure 8.54: Invoke execution phase MSC

To get an impression of the result of this example service, a screenshot of the SIP softphone receiving the SIP MESSAGE request is shown in Figure 8.55. The softphone is called “PhonerLite” (PhonerLite, 2014). In the text window on the right side of the screenshot, the received message “test message” is displayed. The message was received from “testservice@192.168.67.49”. The SIP URI of the user is “sip:testuser@192.168.67.15” (displayed on the status line).



15:23:39: testservice@192.168.67.49 (testservice)
test message

192.168.67.49
sip:testuser@192.168.67.15

Figure 8.55: SIP message received in SIP user agent

8.3.7 Implementation of the Receive LSBB together with HTTP RCSBB

The Receive LSBB can receive events from RCSBBs. The execution of the service waits until the Receive LSBB receives an inter-service event from the corresponding RCSBB.

The example service, which is used for the proof of concept, waits for a HTTP GET request. Therefore, a HTTP CBB was developed. The HTTP CBB supports the HTTP protocol functionalities that are required for this service. The HTTP CBB consists of the HTTP RCSBB, the (JAIN SLEE) HTTP RA, the “HTTPServices” partner link, the HTTP request, and HTTP response data types. When the HTTP RA receives an HTTP GET request, it sends an “onGETEvent” event to the HTTP RCSBB that triggers the Receive LSBB with an inter-service event.

The graphical representation of the example BPEL process is depicted in Figure 8.56. An excerpt of the corresponding XML document is shown in Figure 8.57.

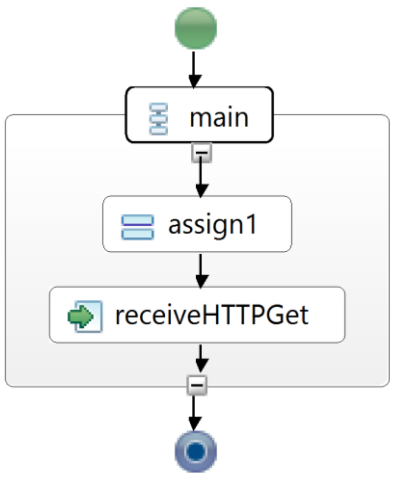
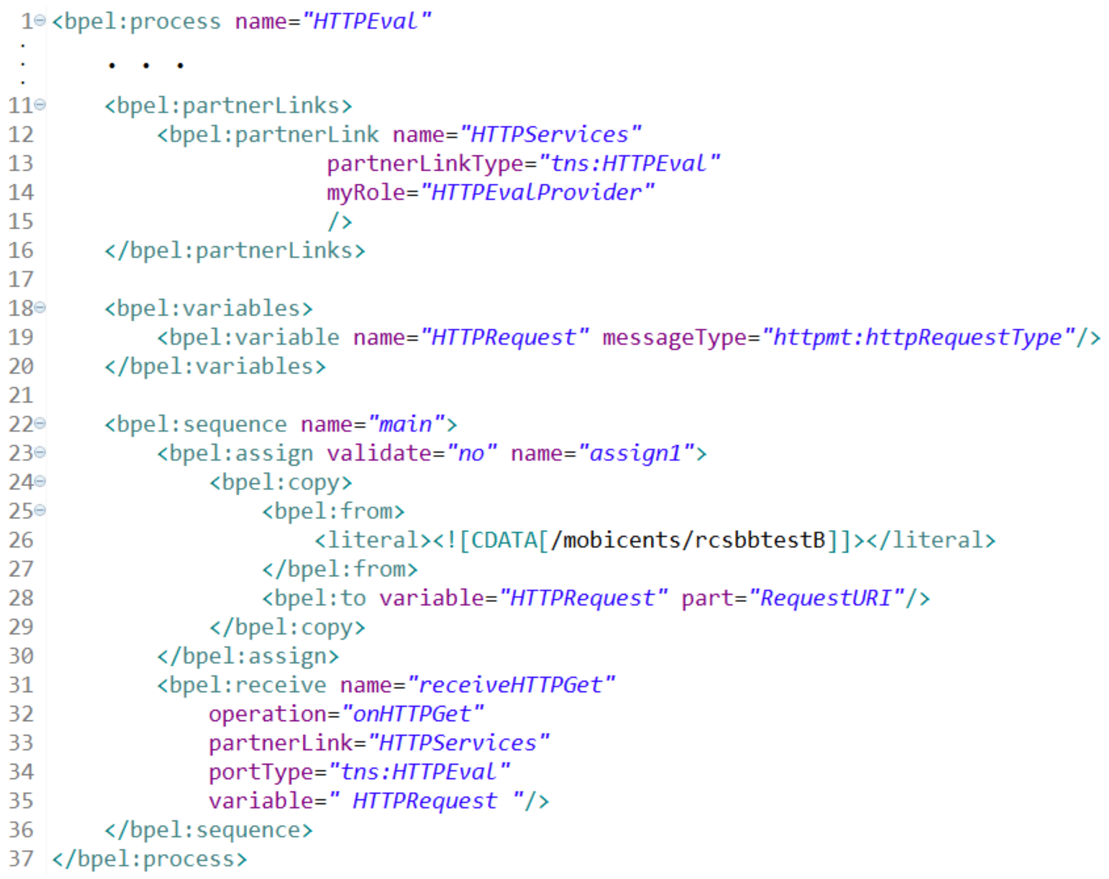


Figure 8.56: Graphical representation of the receive evaluation BPEL process

The name of the BPEL process is “HTTPEval” (line 1). The partner link, which is required for the service, is called “HTTPServices” (line 11 to 16). An HTTP request variable is required to define the URI on which the HTTP RA is listening for incoming HTTP requests. The variable is from the type “httpRequestType” with the name “HTTPRequest” (line 18 to 20).

The BPEL process consists of three activities, a Sequence activity “main” (line 22 to 36) an Assign activity “assign1” (line 23 to 30), and a Receive activity “receiveHTTPGet” (line 31 to 35). The Sequence activity contains the other two activities. The Assign activity consists of one “`copy`” operation (line 24 to 29). Here, the “RequestURI” part of the service variable “HTTPRequest” is set to the request URI “/mobicents/rcsbbtestB”. The Receive LSBB defines the partner link, method, port type, and the variable that should be used in this service.



```
<bpel:process name="HTTPEval"

    . . .

    <bpel:partnerLinks>
        <bpel:partnerLink name="HTTPServices"
                      partnerLinkType="tns:HTTPEval"
                      myRole="HTTPEvalProvider"
                      />
    </bpel:partnerLinks>

    <bpel:variables>
        <bpel:variable name="HTTPRequest" messageType="httpmt:httpRequestType"/>
    </bpel:variables>

    <bpel:sequence name="main">
        <bpel:assign validate="no" name="assign1">
            <bpel:copy>
                <bpel:from>
                    <literal><![CDATA[/mobicents/rcsbbtestB]]></literal>
                </bpel:from>
                <bpel:to variable="HTTPRequest" part="RequestURI"/>
            </bpel:copy>
        </bpel:assign>
        <bpel:receive name="receiveHTTPGet"
            operation="onHTTPGet"
            partnerLink="HTTPServices"
            portType="tns:HTTPEval"
            variable=" HTTPRequest "/>
    </bpel:sequence>
</bpel:process>
```

Figure 8.57: XML document excerpt of the "HTTPEval" process

In the service composition phase it is expected that the LSBBs and the HTTP RCSBB are identified, created, and configured. The composition phase is triggered by a configuration event sent from the framework management to the SCMSBB, which then sends configuration events to the identified service components. When the components have completed their configuration tasks, each of them sends a ready event to the SCMSBB, which then itself sends a "ready event" back to the framework management.

During the execution phase, inter-service events are sent between the components, in the order defined by the service description. The framework management triggers the execution by sending an inter-service event to the SCMSBB. From there an inter-service event is sent to the Sequence LSBB, which sends an inter-service event to the

Assign LSBB. After the request variables have been configured, the HTTP RCSBB is triggered and starts waiting for "onGETEvent" events from the HTTP RA. When the HTTP RA receives an HTTP request, it generates an "onGETEvent" event. This event is sent to the HTTP RCSBB. With this event, the HTTP RCSBB is triggered and executed. It creates an inter-service event that is sent to the Receive LSBB. The Receive LSBB is the last LSBB of the service instance. Therefore, it will send its inter-service event to the SCMSBB, which also sends an inter-service event to the framework management to inform it about the successful execution of the service instance.

The log output of the service composition phase is illustrated in Figure 8.58 and in Figure 8.59. The MSC of the event communication between the components in the composition phase is shown in Figure 8.60. The log output of the service execution phase is given in Figure 8.61. The corresponding MSC with the event communication and the HTTP protocol communication is shown in Figure 8.62.

```
[FrameworkManagementSBB] fire ConfigurationEvent >>
[ServiceControlMSBB] << ConfigurationEvent received
[ServiceControlMSBB] service name:HTTPEval
[ServiceControlMSBB] create new service instance -> parsing description
[ServiceControlMSBB] found imports
[ServiceControlMSBB] found imports
[ServiceControlMSBB] partnerLinks:
[ServiceControlMSBB] partnerLink: -> name: HTTPServices
[ServiceControlMSBB] variables:
[ServiceControlMSBB] Variable: httpRequestType TypeClass: de.rcsbb.http.HttpRequestType
[ServiceControlMSBB] found element -> sequence -> name: main
[ServiceControlMSBB] ElementIdentifier: sequence#HTTPEval_29504287386578#1
[ServiceControlMSBB] fire ConfigurationEvent >> sequence#HTTPEval_29504287386578#1 >>
[ServiceControlMSBB] found element -> assign -> name: Assign1
[ServiceControlMSBB] ElementIdentifier: assign#HTTPEval_29504287386578#3
[ServiceControlMSBB] fire ConfigurationEvent >> assign#HTTPEval_29504287386578#3 >>
[ServiceControlMSBB] found element -> receive -> name: ReceiveHTTPGet
[ServiceControlMSBB] ElementIdentifier: receive#HTTPEval_29504287386578#5
[ServiceControlMSBB] found element -> rcsbb -> name: HTTPServices#HTTPEval_29504287386578#7
[ServiceControlMSBB] fire ConfigurationEvent >> HTTPServices#HTTPEval_29504287386578#7 >>
[ServiceControlMSBB] fire ConfigurationEvent >> receive#HTTPEval_29504287386578#5 >>
```

Figure 8.58: Receive composition evaluation log – part 1

The SCMSBB receives the inter-service event from the framework management and starts analysing the service description. It finds the partner link "HTTPServices" and the “HTTPRequestType” variable. The SCMSBB identifies the required LSBBs and the HTTP RCSBB, and sends configuration events for the creation and configuration of these components (Figure 8.58 and Figure 8.60).

```
[Sequence] << ConfigurationEvent received sequence#HTTPEval_29504287386578#1
[Sequence] fire ReadyEvent >> sequence#HTTPEval_29504287386578#1 >>
[ServiceControlMSBB] << ReadyEvent received! << sequence#HTTPEval_29504287386578#1
[ServiceControlMSBB] Waiting for elements to become ready!
[Assign] << ConfigurationEvent received assign#HTTPEval_29504287386578#3
[Assign] fireReadyEvent >> assign#HTTPEval_29504287386578#3 >>
[ServiceControlMSBB] << ReadyEvent received! << assign#HTTPEval_29504287386578#3
[ServiceControlMSBB] Waiting for elements to become ready!
[HTTP_RCSBB] << onConfigurationEvent received <<
[HTTP_RCSBB] fire ReadyEvent >>
[ServiceControlMSBB] << ReadyEvent received! << HTTPServices#HTTPEval_29504287386578#7
[ServiceControlMSBB] Waiting for elements to become ready!
[Receive] ->onConfigurationEventnull
[Receive] fireReadyEvent->receive#HTTPEval_29504287386578#5
[ServiceControlMSBB] << ReadyEvent received! << receive#HTTPEval_29504287386578#5
[ServiceControlMSBB] fire ReadyEvent >> HTTPEval_29504287386578
[FrameworkManagementSBB] << ReadyEvent received << HTTPEval_29504287386578
```

Figure 8.59: Receive composition evaluation log – part 2

After the components have finished their configuration, they send ready events back to the SCMSBB (Figure 8.59 and Figure 8.60). The SCMSBB waits until all ready events have been received, and then it sends an own ready event to the framework management. Now, the service is created, configured, and ready for execution.

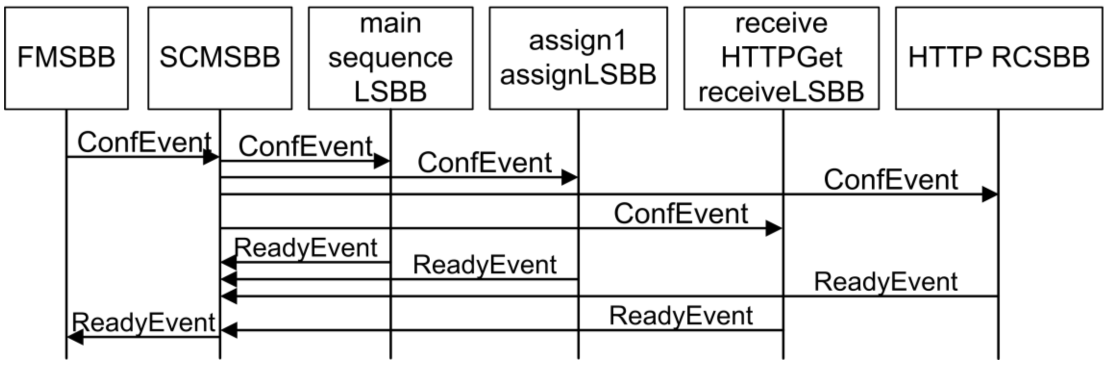
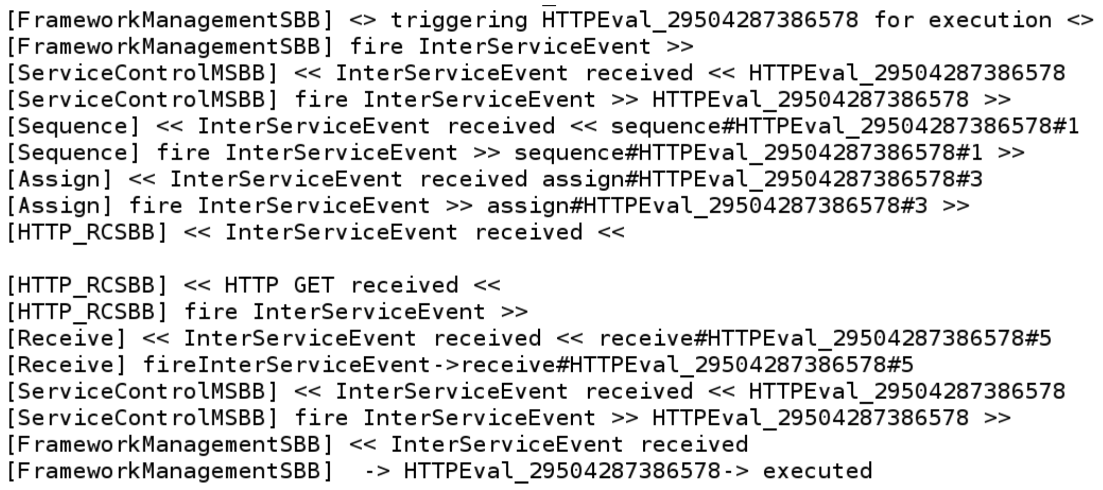


Figure 8.60: Receive composition phase MSC

The execution phase starts with an inter-service event from the framework management to the SCMSBB (Figure 8.61 and Figure 8.62). The components are executed in the expected order. After the Assign LSBB has configured the request

URI in the HTTP request variable, it will fire an inter-service event to the HTTP RCSBB. The RCSBB waits until it receives an “onGETEvent” from the HTTP RA with the defined request URI, then it sends an inter-service event to the Receive LSBB. This LSBB is the last LSBB of the service instance. It sends an inter-service event back to the SCMSBB. To confirm the successful execution of the service instance, the SCMSBB informs the framework management by sending an inter-service event.



```
[FrameworkManagementSBB] <> triggering HTTPEval_29504287386578 for execution <>
[FrameworkManagementSBB] fire InterServiceEvent >>
[ServiceControlMSBB] << InterServiceEvent received << HTTPEval_29504287386578
[ServiceControlMSBB] fire InterServiceEvent >> HTTPEval_29504287386578 >>
[Sequence] << InterServiceEvent received << sequence#HTTPEval_29504287386578#1
[Sequence] fire InterServiceEvent >> sequence#HTTPEval_29504287386578#1 >>
[Assign] << InterServiceEvent received assign#HTTPEval_29504287386578#3
[Assign] fire InterServiceEvent >> assign#HTTPEval_29504287386578#3 >>
[HTTP_RCSBB] << InterServiceEvent received <<

[HTTP_RCSBB] << HTTP GET received <<
[HTTP_RCSBB] fire InterServiceEvent >>
[Receive] << InterServiceEvent received << receive#HTTPEval_29504287386578#5
[Receive] fireInterServiceEvent->receive#HTTPEval_29504287386578#5
[ServiceControlMSBB] << InterServiceEvent received << HTTPEval_29504287386578
[ServiceControlMSBB] fire InterServiceEvent >> HTTPEval_29504287386578 >>
[FrameworkManagementSBB] << InterServiceEvent received
[FrameworkManagementSBB]  -> HTTPEval_29504287386578-> executed
```

Figure 8.61: Receive execution evaluation log

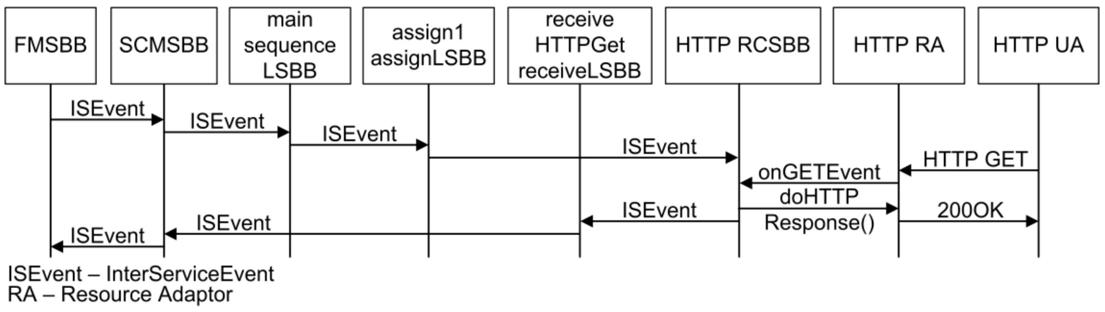


Figure 8.62: Receive execution phase MSC

8.3.8 Implementation of the Wait LSBB

The Wait activity offers the possibility to define a delay before the next activity is executed. The Wait activity is completed when the specified deadline or duration is reached. This activity is required for the test scenario in section 8.4.

For the evaluation of the Wait activity, a minimal example BPEL process has been developed. The graphical representation of the process is given in Figure 8.63, and the XML document representation in Figure 8.64.

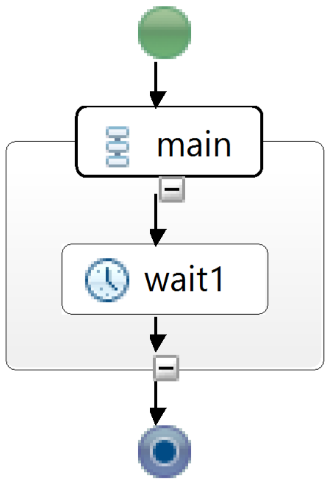
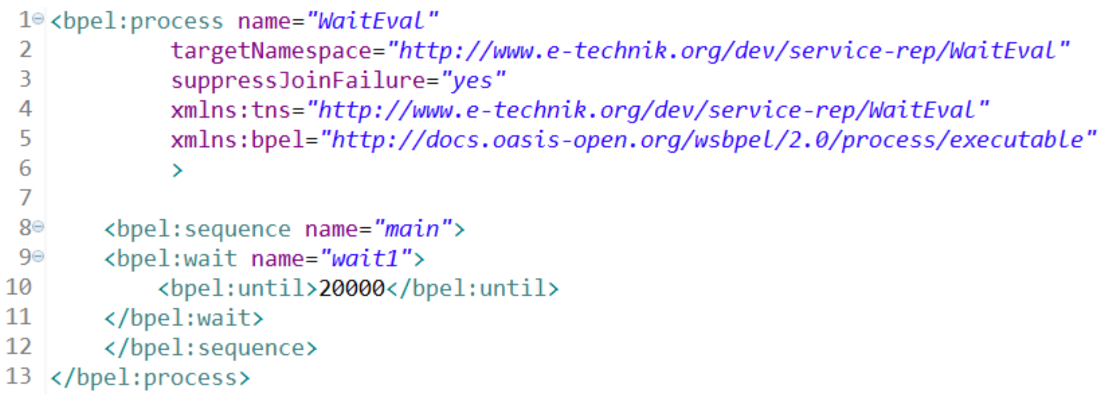


Figure 8.63: Graphical representation of the wait evaluation BPEL process

The name of the BPEL process is “WaitEval” (line 1). It consists of only two activities, the Sequence activity “main” (line 8 to 12), and the Wait activity “wait1” (line 9 to 11). The Wait activity is contained within the Sequence activity. The duration of the Wait activity is set to 20000 milliseconds (line 10).



```
<bpel:process name="WaitEval"
        targetNamespace="http://www.e-technik.org/dev/service-rep/WaitEval"
        suppressJoinFailure="yes"
        xmlns:tns="http://www.e-technik.org/dev/service-rep/WaitEval"
        xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
        >

    <bpel:sequence name="main">
    <bpel:wait name="wait1">
        <bpel:until>20000</bpel:until>
    </bpel:wait>
    </bpel:sequence>
</bpel:process>
```

Figure 8.64: XML document of the "WaitEval" process

It is assumed that the service instance has been created and configured in the composition phase. In the execution phase, the service is executed until the Wait LSBB is reached. Then the Wait LSBB starts the timer and waits 20000 milliseconds until the timer is expired. A timer event then activates the Wait LSBB again, and the execution of the service instance continues.

The log output of the framework during the service composition phase is given in Figure 8.65, and the MSC of the event communication between the service components is shown in Figure 8.66.

```
[FrameworkManagementSBB] fire ConfigurationEvent >>
[ServiceControlMSBB] << ConfigurationEvent received -> #ID: WaitEval_24811136756290
[ServiceControlMSBB] service name:WaitEval
[ServiceControlMSBB] create new service instance -> parsing description
[ServiceControlMSBB] found element -> sequence -> name: main
[ServiceControlMSBB] ElementIdentifier: sequence#WaitEval_24900802361491#6
[ServiceControlMSBB] fire ConfigurationEvent >> sequence#WaitEval_24900802361491#6 >>
[ServiceControlMSBB] found element -> wait -> name: wait1
[ServiceControlMSBB] ElementIdentifier: wait#WaitEval_24900802361491#8
[ServiceControlMSBB] fire ConfigurationEvent >> wait#WaitEval_24900802361491#8 >>
[Sequence] << ConfigurationEvent received sequence#WaitEval_24900802361491#6
[Sequence] fire ReadyEvent >> sequence#WaitEval_24900802361491#6 >>
[ServiceControlMSBB] << ReadyEvent received! << sequence#WaitEval_24900802361491#6
[ServiceControlMSBB] Waiting for elements to become ready!
[Wait] << ConfigurationEvent received wait#WaitEval_24900802361491#8
[Wait] fire ReadyEvent >> wait#WaitEval_24900802361491#8 >>
[ServiceControlMSBB] << ReadyEvent received! << wait#WaitEval_24900802361491#8
[ServiceControlMSBB] fire ReadyEvent >> WaitEval_24900802361491
[FrameworkManagementSBB] << ReadyEvent received << WaitEval_24900802361491
[FrameworkManagementSBB] <---> WaitEval_24900802361491 created and configured <--->
```

Figure 8.65: Wait composition evaluation log

As expected, the composition phase is triggered by the framework management. The description is parsed by the SCMSBB, and the creation and configuration of the Sequence LSBB and of the Wait LSBB is triggered by configuration events. Both LSBBs answer with a ready event and the SCMSBB confirms with a ready event to the framework management that the service instance has been created.

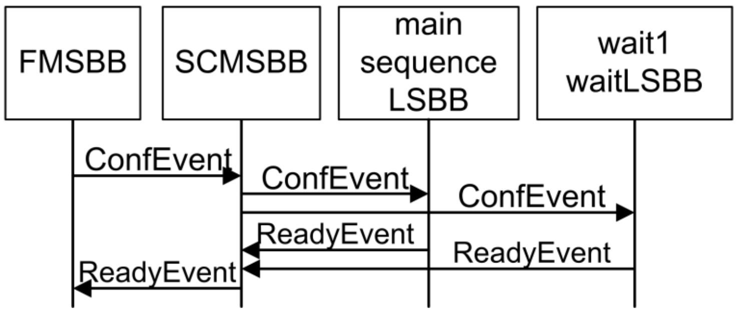
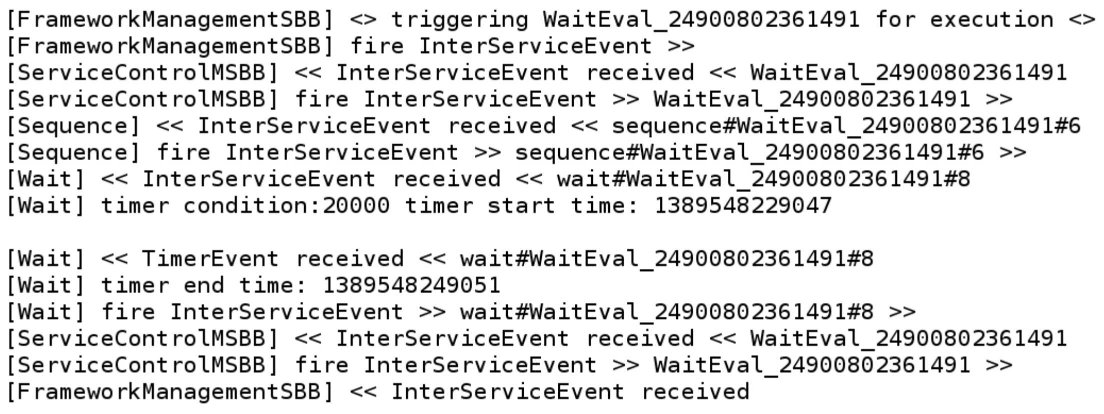


Figure 8.66: Wait composition phase MSC

The log output of the execution phase is shown in Figure 8.67, and the event communication during the execution phase in the MSC of Figure 8.68.

To evaluate the Wait LSBB, some log output was added. This output consists of a timestamp shortly before the Wait LSBB starts the timer, and a timestamp shortly after the timer event has occurred.



```
[FrameworkManagementSBB] <> triggering WaitEval_24900802361491 for execution <>
[FrameworkManagementSBB] fire InterServiceEvent >>
[ServiceControlMSBB] << InterServiceEvent received << WaitEval_24900802361491
[ServiceControlMSBB] fire InterServiceEvent >> WaitEval_24900802361491 >>
[Sequence] << InterServiceEvent received << sequence#WaitEval_24900802361491#6
[Sequence] fire InterServiceEvent >> sequence#WaitEval_24900802361491#6 >>
[Wait] << InterServiceEvent received << wait#WaitEval_24900802361491#8
[Wait] timer condition:20000 timer start time: 1389548229047

[Wait] << TimerEvent received << wait#WaitEval_24900802361491#8
[Wait] timer end time: 1389548249051
[Wait] fire InterServiceEvent >> wait#WaitEval_24900802361491#8 >>
[ServiceControlMSBB] << InterServiceEvent received << WaitEval_24900802361491
[ServiceControlMSBB] fire InterServiceEvent >> WaitEval_24900802361491 >>
[FrameworkManagementSBB] << InterServiceEvent received
```

Figure 8.67: Wait execution evaluation log

The execution starts with an inter-service event from the framework management. Each LSBB in the workflow is executed until the Wait LSBB is reached. The Wait LSBB prints out the timestamp (1389548229047 ms), starts the timer (20000 ms), and waits until the timer ends. When the timer event occurs, a new timestamp is printed out (1389548249051 ms). The difference (1389548249051 ms - 1389548229047 ms = 20004 ms) confirms the expected value. The addition of 4 ms is caused by timer setup, event handling, and the print instruction. The Wait LSBB is the last service component of the service; it sends an inter-service event to the SCMSBB, and the SCMSBB confirms the service instance execution to the framework management with an inter-service event.

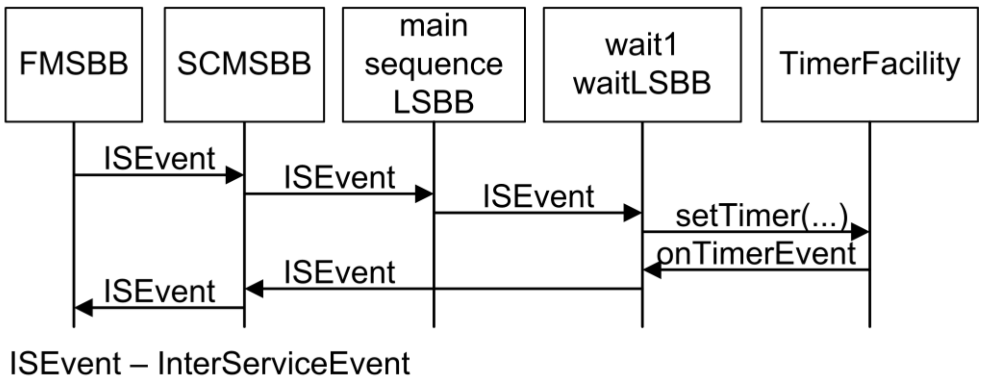


Figure 8.68: Wait execution phase MSC

This section discussed the implementation of the Wait LSBB and demonstrated how the Wait activity can be used in order to realise the wait functionality.

8.3.9 Implementation of the While LSBB

A possibility to define loops in BPEL is the While activity. The While LSBB is the equivalent of the While activity in the SEE and has been implemented in the prototype. The condition of the While activity is evaluated for each loop. If the condition evaluates to true, the activities contained within the While activity are executed. If the condition evaluates to false, then the next activity after the While activity is executed.

The While LSBB is evaluated with the help of an example BPEL process. The graphical representation of the process is shown in Figure 8.69, and the XML document of this process is depicted in Figure 8.70 and Figure 8.71.

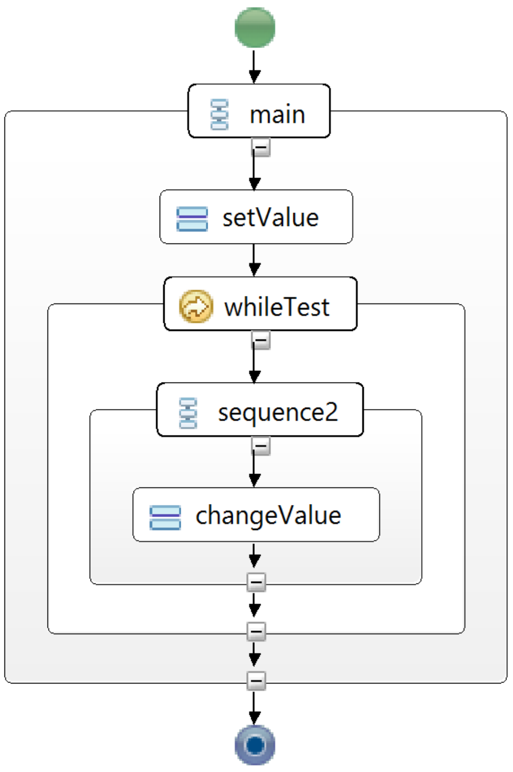
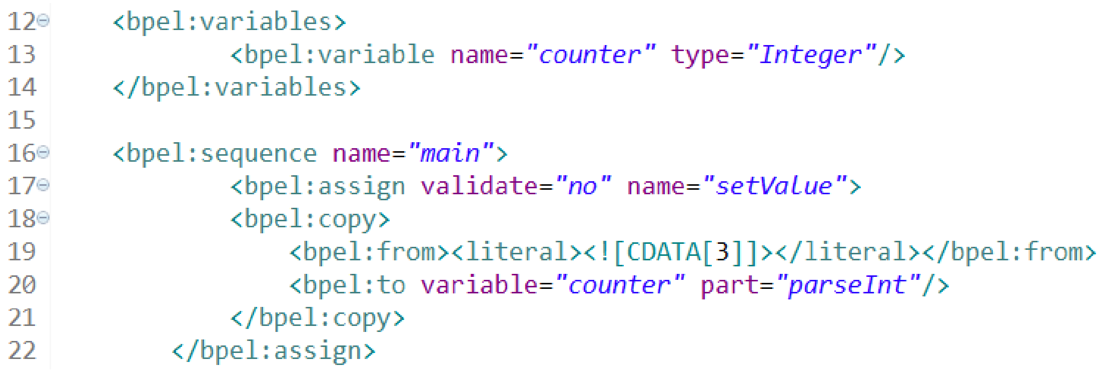


Figure 8.69: Graphical representation of the while evaluation BPEL process

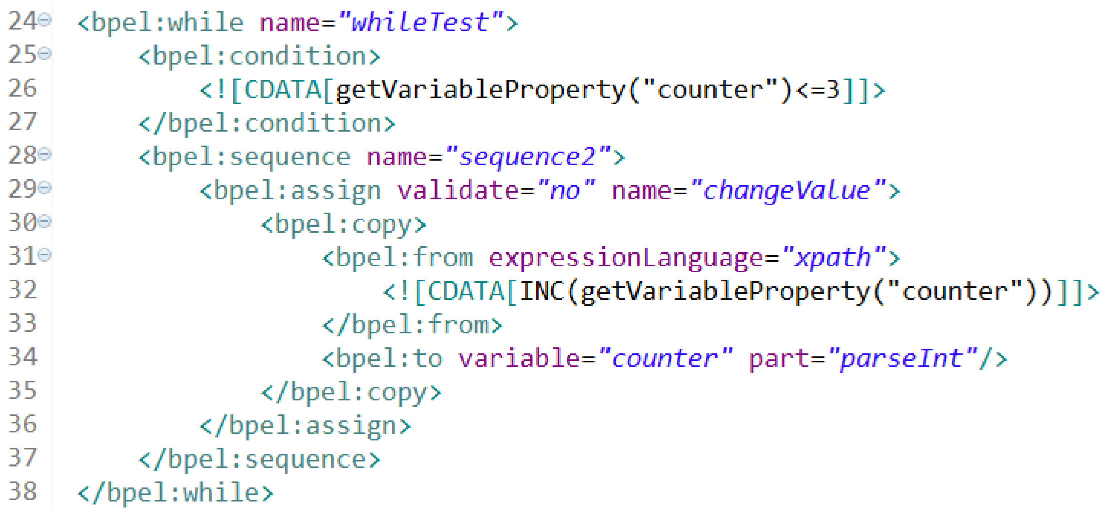
The BPEL process "WhileEval" consists of five activities, two sequences, a while loop, and two assigns. The integer variable "counter" is required to demonstrate the behaviour of the while condition.



```
12   <bpel:variables>
13           <bpel:variable name="counter" type="Integer"/>
14   </bpel:variables>
15
16   <bpel:sequence name="main">
17           <bpel:assign validate="no" name="setValue">
18           <bpel:copy>
19               <bpel:from><literal><![CDATA[3]]></literal></bpel:from>
20               <bpel:to variable="counter" part="parseInt"/>
21           </bpel:copy>
22       </bpel:assign>
```

Figure 8.70: XML document excerpt of the “WhileEval” process – part 1

The first activity of the process is the Sequence activity “main” (Figure 8.70, line 16), which contains all other activities. The first activity within the sequence is the Assign activity “setValue” (Figure 8.70, line 17 to 22). The “`copy`” operation of the assign sets the value of the counter variable to “3”. This value is required for the condition of the While activity.



```
24 <bpel:while name="whileTest">
25     <bpel:condition>
26         <![CDATA[getVariableProperty("counter")<=3]]>
27     </bpel:condition>
28     <bpel:sequence name="sequence2">
29         <bpel:assign validate="no" name="changeValue">
30             <bpel:copy>
31                 <bpel:from expressionLanguage="xpath">
32                     <![CDATA[INC(getVariableProperty("counter"))]]>
33                 </bpel:from>
34                 <bpel:to variable="counter" part="parseInt"/>
35             </bpel:copy>
36         </bpel:assign>
37     </bpel:sequence>
38 </bpel:while>
```

Figure 8.71: XML document excerpt of the “WhileEval” process – part 2

The name of the While activity is “whileTest” (Figure 8.71, line 24 to 38). The while condition is evaluated to “true”, if the counter variable is lower than or equal (<=) to “3”. As described in section 8.2, some example operations, expressions, and conditions are supported by the prototype. To read out the value of the variable "counter", the "getVariableProperty" operation is used. The While activity contains the Sequence activity "sequence2" (Figure 8.71, line 28 to 37), which, again, contains the Assign activity "changeValue" (Figure 8.71, line 29 to 36). The research prototype requires a Sequence activity within the While activity. Activities that should be added to the while loop must be added to this Sequence activity. In the "copy" operation (line 30 to 35) within the Assign activity, the value of the variable "counter" is read with the "getVariableProperty" operation and incremented with the "INC" operation.

In the configuration phase, it is expected that the SCMSBB starts analysing the service description when it receives a configuration event from the framework management. It creates the integer variable "counter" and identifies the required LSBBs. The creation and configuration of the LSBBs is triggered with configuration events. Upon configuration of components, they should answer with ready events. The SCMSBB waits for the ready events from the LSBBs and sends a ready event to the framework management to confirm the creation and configuration of the service instance.

In the execution phase, the SCMSBB receives an inter-service event from the service management. The SCMSBB sends an event to the first LSBB of the service, the Sequence LSBB. This LSBB activates the Assign LSBB "setValue" to set the value of the variable "counter" to "3". Then the While LSBB starts its execution. In the first step, the condition is evaluated. The value of the variable "counter" is "3"; therefore, the condition evaluates to "true" (3 <= 3), and the components contained within the loop are executed. The Sequence LSBB "sequence2" receives an event from the While LSBB and executes the contained LSBBs in a sequential order. The only component within the “sequence2” is the Assign LSBB “changeValue”. The “copy” operation of the Assign LSBB increments the value of the variable “counter” to “4”.

Now, the components within the while loop are executed once, and the condition is checked again. The value of the variable “counter” is “4”, which is why the condition should evaluate to “false” (4 <= 3) and the While LSBB sends an inter-service event to the next component after the loop. The While LSBB is the last LSBB of the service instance; therefore, the event is sent back to the SCMSBB, and the SCMSBB confirms the execution of the service instance to the framework management with an inter-service event. The service description is uploaded to the service repository and triggered by the user for composition and execution through the interactive web interface. The log output of the composition phase is given in Figure 8.72 and Figure 8.73. The MSC with the event communication during the composition phase is depicted in Figure 8.74. The log output of the execution phase is illustrated in Figure 8.75, and the correspondent MSC with the event communication during the execution phase is shown in Figure 8.76.

The framework management starts the composition phase with a configuration event for the SCMSBB. As expected, the service description is analysed, the variable is found, and configuration events are sent to the required LSBBs. The LSBBs return a ready event to the SCMSBB when they are created and configured. Then the SCMSBB confirm the creation and configuration of the service instance to the framework management with a ready event.

```
[FrameworkManagementSBB] fire ConfigurationEvent >>
[ServiceControlMSBB] << ConfigurationEvent received
[ServiceControlMSBB] service name:WhileEval
[ServiceControlMSBB] create new service instance -> parsing description
[ServiceControlMSBB] found imports
[ServiceControlMSBB] found imports
[ServiceControlMSBB] partnerLinks:
[ServiceControlMSBB] variables:
[ServiceControlMSBB] Variable: Integer TypeClass: java.lang.Integer
[ServiceControlMSBB] found element -> sequence -> name: main
[ServiceControlMSBB] ElementIdentifier: sequence#WhileEval_4335962461836#1
[ServiceControlMSBB] fire ConfigurationEvent >> sequence#WhileEval_4335962461836#1 >>
[ServiceControlMSBB] found element -> assign -> name: setValue
[ServiceControlMSBB] ElementIdentifier: assign#WhileEval_4335962461836#3
[ServiceControlMSBB] fire ConfigurationEvent >> assign#WhileEval_4335962461836#3 >>
[ServiceControlMSBB] found element -> while -> name: whileTest
[ServiceControlMSBB] ElementIdentifier: while#WhileEval_4335962461836#5
[ServiceControlMSBB] found element -> sequence -> name: sequence2
[ServiceControlMSBB] ElementIdentifier: sequence#WhileEval_4335962461836#8
[ServiceControlMSBB] fire ConfigurationEvent >> sequence#WhileEval_4335962461836#8 >>
[ServiceControlMSBB] found element -> assign -> name: changeValue
[ServiceControlMSBB] ElementIdentifier: assign#WhileEval_4335962461836#10
[ServiceControlMSBB] fire ConfigurationEvent >> assign#WhileEval_4335962461836#10 >>
[ServiceControlMSBB] fire ConfigurationEvent >> while#WhileEval_4335962461836#5 >>
```

Figure 8.72: While composition evaluation log – part 1

```
[Sequence] << ConfigurationEvent received sequence#WhileEval_4335962461836#1
[Sequence] fire ReadyEvent >> sequence#WhileEval_4335962461836#1 >>
[ServiceControlMSBB] << ReadyEvent received! << sequence#WhileEval_4335962461836#1
[ServiceControlMSBB] Waiting for elements to become ready!
[Assign] << ConfigurationEvent received assign#WhileEval_4335962461836#3
[Assign] fireReadyEvent >> assign#WhileEval_4335962461836#3 >>
[ServiceControlMSBB] << ReadyEvent received! << assign#WhileEval_4335962461836#3
[ServiceControlMSBB] Waiting for elements to become ready!
[Sequence] << ConfigurationEvent received sequence#WhileEval_4335962461836#8
[Sequence] fire ReadyEvent >> sequence#WhileEval_4335962461836#8 >>
[ServiceControlMSBB] << ReadyEvent received! << sequence#WhileEval_4335962461836#8
[ServiceControlMSBB] Waiting for elements to become ready!
[Assign] << ConfigurationEvent received assign#WhileEval_4335962461836#10
[Assign] fireReadyEvent >> assign#WhileEval_4335962461836#10 >>
[ServiceControlMSBB] << ReadyEvent received! << assign#WhileEval_4335962461836#10
[ServiceControlMSBB] Waiting for elements to become ready!
[While] << ConfigurationEvent received << while#WhileEval_4335962461836#5
[While] fire ReadyEvent >> while#WhileEval_4335962461836#5
[ServiceControlMSBB] << ReadyEvent received! << while#WhileEval_4335962461836#5
[ServiceControlMSBB] fire ReadyEvent >> WhileEval_4335962461836
[FrameworkManagementSBB] << ReadyEvent received << WhileEval_4335962461836
[FrameworkManagementSBB] <---> WhileEval_4335962461836 created and configured <--->
```

Figure 8.73: While composition evaluation log – part 2

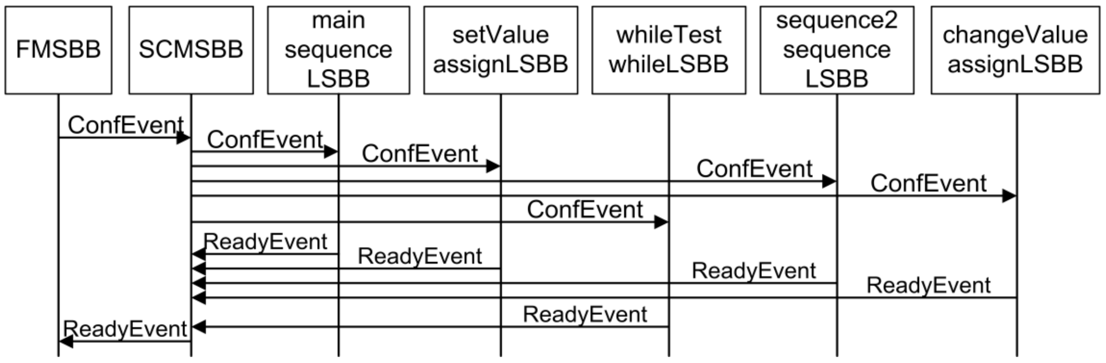
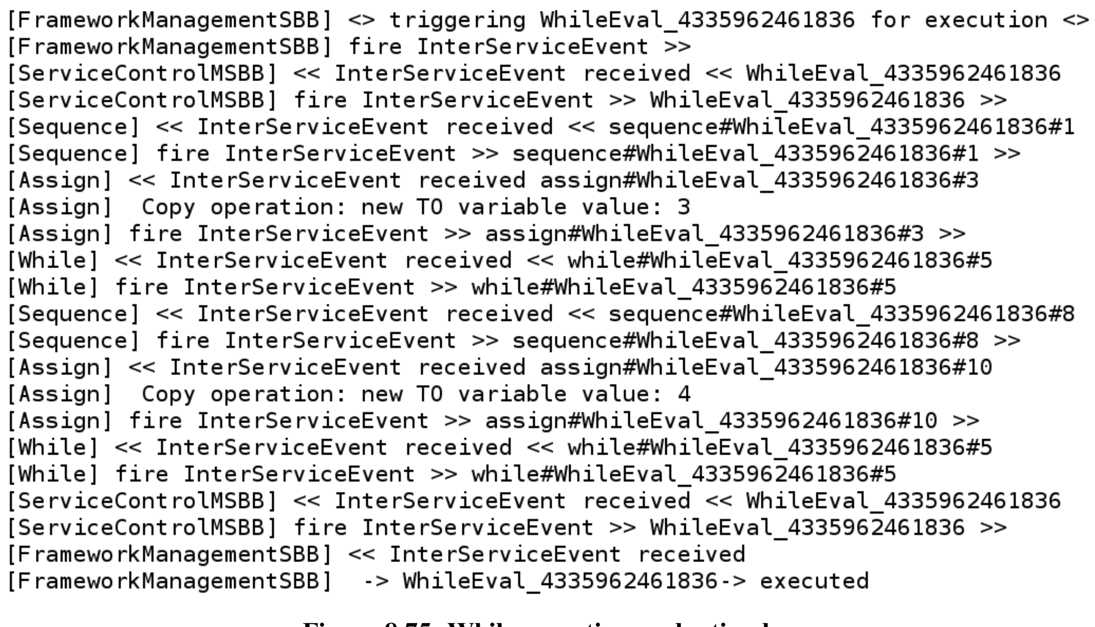


Figure 8.74: While composition phase MSC

The framework management triggers the execution phase by sending an inter-service event to the SCMSBB. The workflow is executed as expected. The Sequence LSBB "main" is the first LSBB that is executed. It triggers the Assign LSBB "setValue", which sets the value of the variable "counter" to "3". The next component is the While LSBB. As expected, the while loop is executed, because the condition was evaluated to "true". The Sequence LSBB "sequence2" and the Assign LSBB "changeValue" within the loop are executed, and the variable "counter" is increased to "4".

The Assign LSBB sends an inter-service event back to the While LSBB, and the condition is evaluated again. Now, the value of the variable "counter" is "4". The while loop is not executed anymore because the condition (4 <= 3) is evaluated to "false", the While LSBB fires an inter-service event back to the SCMSBB. The framework management receives an inter-service event from the SCMSBB as a confirmation that the service has been executed.



```
[FrameworkManagementSBB] <> triggering WhileEval_4335962461836 for execution <>
[FrameworkManagementSBB] fire InterServiceEvent >>
[ServiceControlMSBB] << InterServiceEvent received << WhileEval_4335962461836
[ServiceControlMSBB] fire InterServiceEvent >> WhileEval_4335962461836 >>
[Sequence] << InterServiceEvent received << sequence#WhileEval_4335962461836#1
[Sequence] fire InterServiceEvent >> sequence#WhileEval_4335962461836#1 >>
[Assign] << InterServiceEvent received assign#WhileEval_4335962461836#3
[Assign]  Copy operation: new TO variable value: 3
[Assign] fire InterServiceEvent >> assign#WhileEval_4335962461836#3 >>
[While] << InterServiceEvent received << while#WhileEval_4335962461836#5
[While] fire InterServiceEvent >> while#WhileEval_4335962461836#5
[Sequence] << InterServiceEvent received << sequence#WhileEval_4335962461836#8
[Sequence] fire InterServiceEvent >> sequence#WhileEval_4335962461836#8 >>
[Assign] << InterServiceEvent received assign#WhileEval_4335962461836#10
[Assign]  Copy operation: new TO variable value: 4
[Assign] fire InterServiceEvent >> assign#WhileEval_4335962461836#10 >>
[While] << InterServiceEvent received << while#WhileEval_4335962461836#5
[While] fire InterServiceEvent >> while#WhileEval_4335962461836#5
[ServiceControlMSBB] << InterServiceEvent received << WhileEval_4335962461836
[ServiceControlMSBB] fire InterServiceEvent >> WhileEval_4335962461836 >>
[FrameworkManagementSBB] << InterServiceEvent received
[FrameworkManagementSBB]  -> WhileEval_4335962461836-> executed
```

Figure 8.75: While execution evaluation log

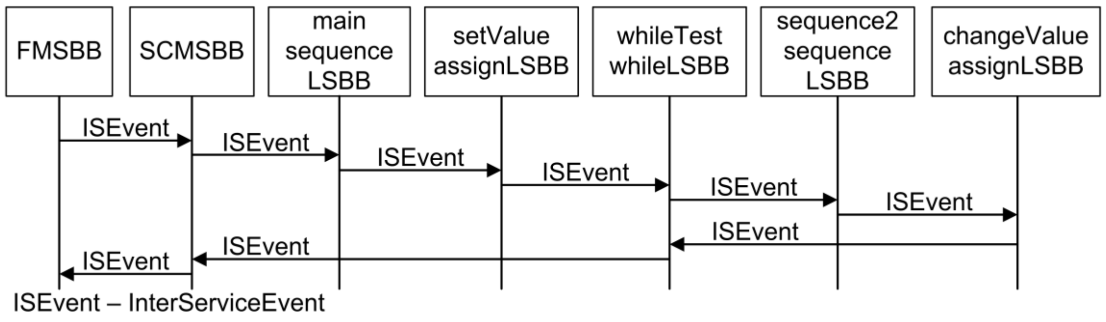


Figure 8.76: While execution phase MSC

8.4 Proof of the Proposed Framework Concept

In order to prove the novel concept proposed in this thesis for automatic service generation from formal service descriptions, the whole prototype framework has to be evaluated. It has to be examined if the proposed example services can be developed with the help of the service description language and the CBBs. The services have to be automatically generated from the service descriptions, and executed within the SEE. It has to be shown that new services can be developed with the framework.

8.4.1 Wake-up Test Scenario

For the proof of concept, an appropriate test scenario is defined. This test scenario realises a typical example of a value-added service, the Wake-up service. This scenario is similar to a conventional service scenario (Martens, 2011) defined for JAIN SLEE mobicents (Mobicents, 2014). The conventional mobicents SIP Wake-up scenario is designed as tutorial how to develop JAIN SLEE services with Java for mobicents. This tutorial already shows the complexity of developing a conventional value-added service for JAIN SLEE. Advanced knowledge in Java, a consolidated understanding of the SIP protocol, and XML knowledge for defining the descriptor files is required. The source code of the conventional JAIN SLEE SBB has more than 400 lines, and several XML descriptor files have to be defined. A development of such a service with conventional service development would certainly take some days.

In this section, the Wake-up service is developed with the research prototype. An overview of the scenario is shown in Figure 8.77.

The service user can send a SIP MESSAGE defining a waiting time in milliseconds. After this duration, a Wake-up SIP MESSAGE will notify the user. SIP has been chosen as an example application protocol. To complete the SIP transactions, the SIP request messages are answered by 200 OK SIP responses.

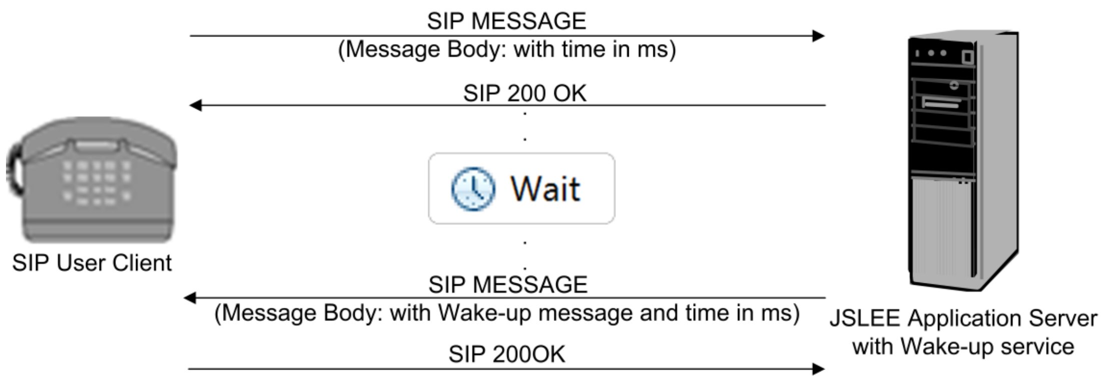


Figure 8.77: Wake-up scenario

The service instance in the application server analyses the message body of the received SIP MESSAGE request and reads out the waiting duration in milliseconds. With this duration a timer is started. Upon timer expiry, the service generates and sends a Wake-up SIP MESSAGE request with waiting duration in milliseconds from the received SIP MESSAGE.

For this scenario, the SIP CBB is used. The service developer does not need detailed knowledge of the SIP protocol. The SIP CBB performs the required protocol specific tasks and hides the complexity of the protocol from the developer.

8.4.2 Describing the Wake-up Service in BPEL

The service designer uses a BPEL development tool to develop the service description. The graphical representation of the BPEL process is shown in Figure 8.78. This process was developed with the Eclipse BPEL Designer (Eclipse 2013).

The XML description of the BPEL process is shown in Figure 8.79, Figure 8.80, and in Figure 8.81.

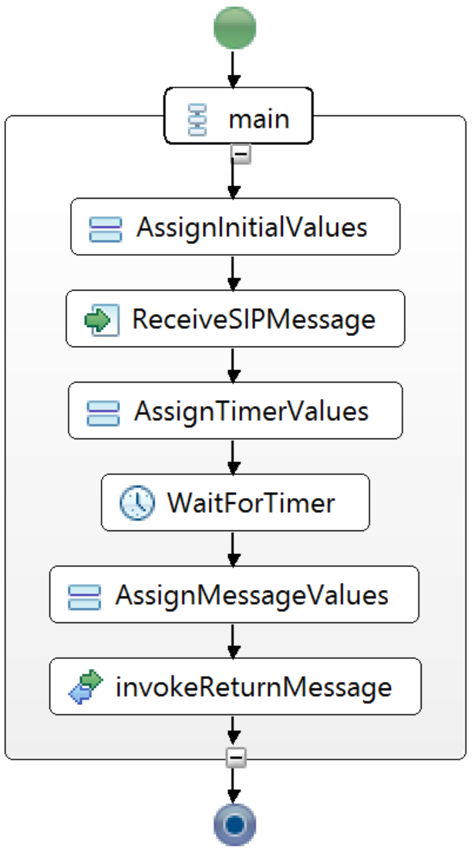
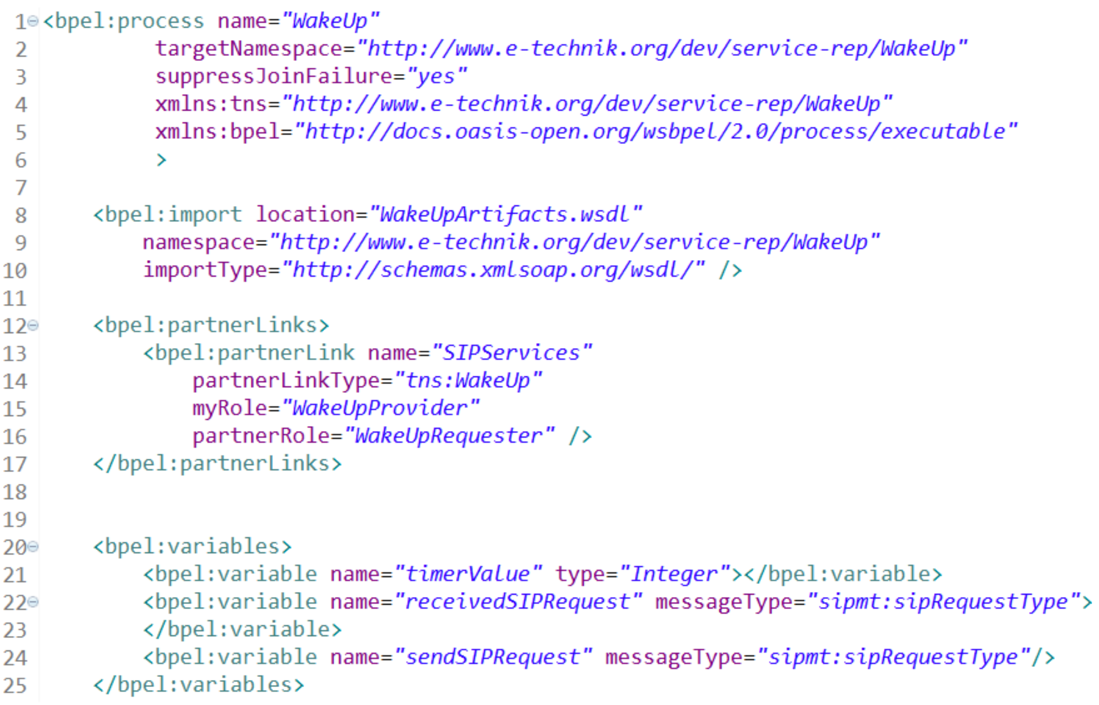


Figure 8.78: Graphical representation of the Wake-up BPEL process

The process consists of seven activities: one Sequence activity (refer to section 8.3.2), three assign activities (refer to section 8.3.3), one Receive activity (refer to section 8.3.7), one Wait activity (refer to section 8.3.8), and one Invoke activity (refer to section 8.3.6). The Sequence activity encapsulates all other activities and describes a sequential execution order from the top to the bottom of the process.

The name of the process is “WakeUp” (Figure 8.79, line 1). As already mentioned, the SIP CBB is used for this service. The name of the correspondent partner link for the SIP CBB is “SIPServices” (Figure 8.79, line 12 to 17). Three variables are required for this service: (i) one integer variable “timerValue” to store the timer value, (ii) one “SIPRequestType” variable “receivedSIPRequest” to store the received SIP request, and (iii) one “SIPRequestType” variable “sendSIPRequest” for the outgoing SIP Message request (Figure 8.79, line 20 to 25).



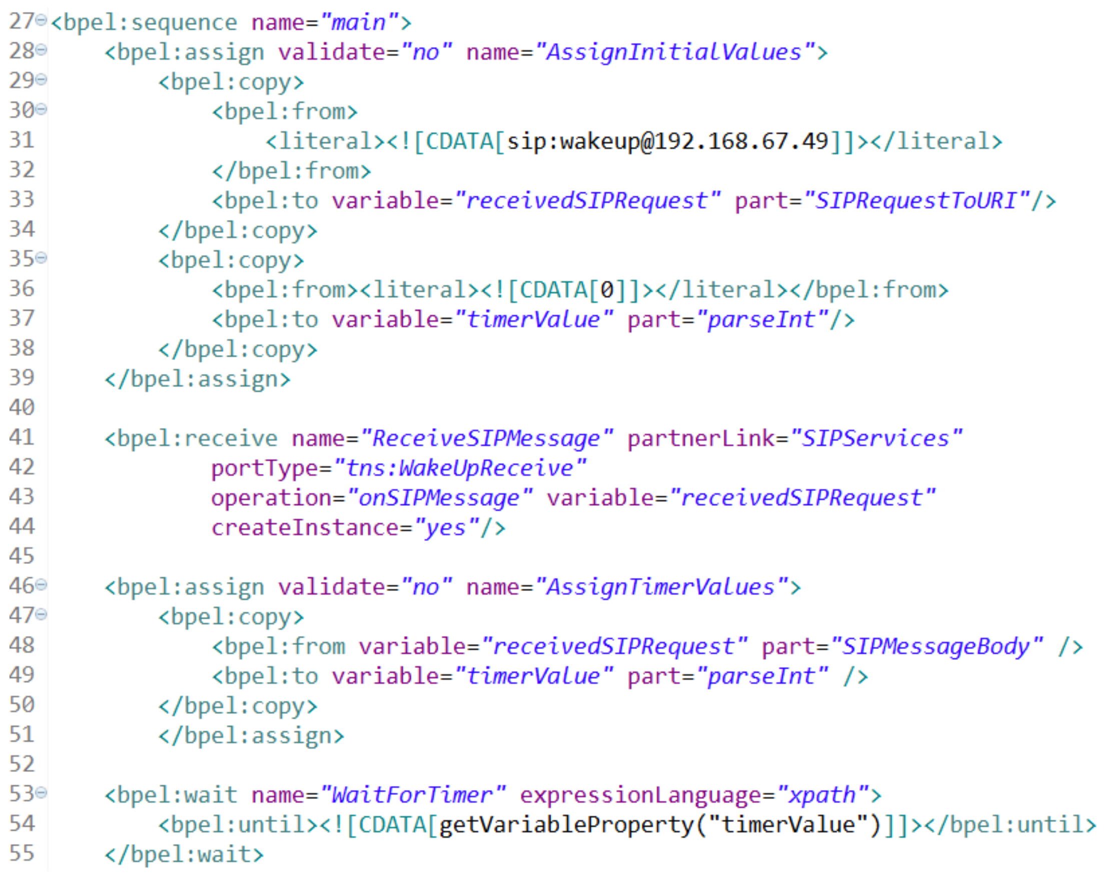
```
 1 <bpel:process name="WakeUp"
 2          targetNamespace="http://www.e-technik.org/dev/service-rep/WakeUp"
 3          suppressJoinFailure="yes"
 4          xmlns:tns="http://www.e-technik.org/dev/service-rep/WakeUp"
 5          xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
 6          >
 7 
 8     <bpel:import location="WakeUpArtifacts.wsdl"
 9         namespace="http://www.e-technik.org/dev/service-rep/WakeUp"
10         importType="http://schemas.xmlsoap.org/wsdl/" />
11 
12     <bpel:partnerLinks>
13         <bpel:partnerLink name="SIPServices"
14             partnerLinkType="tns:WakeUp"
15             myRole="WakeUpProvider"
16             partnerRole="WakeUpRequester" />
17     </bpel:partnerLinks>
18 
19 
20     <bpel:variables>
21         <bpel:variable name="timerValue" type="Integer"></bpel:variable>
22         <bpel:variable name="receivedSIPRequest" messageType="sipmt:sipRequestType">
23         </bpel:variable>
24         <bpel:variable name="sendSIPRequest" messageType="sipmt:sipRequestType"/>
25     </bpel:variables>
```

Figure 8.79: XML document excerpt of the “Wake-up” process – part 1

The Sequence activity with the name "main" includes all other activities (Figure 8.80, line 27; Figure 8.81, line 89). The first activity within the sequence is an Assign activity with the name "AssignInitialValues", which consist of two "`copy`" operations (Figure 8.80, line 28 to 39). In the first "`copy`" operation, the SIP request URI (sip:wakeup@192.168.67.49) is stored in the "SIPRequestToURI" part of the "receivedSIPRequest" variable. SIP requests for this SIP URI will be handled by the Wake-up service. The second "`copy`" operation initiates the "timerValue" variable with "0".

The next activity in the service description is the Receive activity "ReceiveSIPMessage" (Figure 8.80, line 41 to 44). Here it is defined that the "`onSIPMessage`" operation of the "SIPServices" partner link is used within this activity. The Receive activity uses the "receivedSIPRequest" variable to define the SIP URI on which the service is listening for incoming SIP MESSAGES and on which the SIP RCSBB will be triggered within the service execution phase. In this example service the SIP URI is "sip:wakeup@192.168.67.49". The received SIP request is also stored within the "receivedSIPRequest" variable.

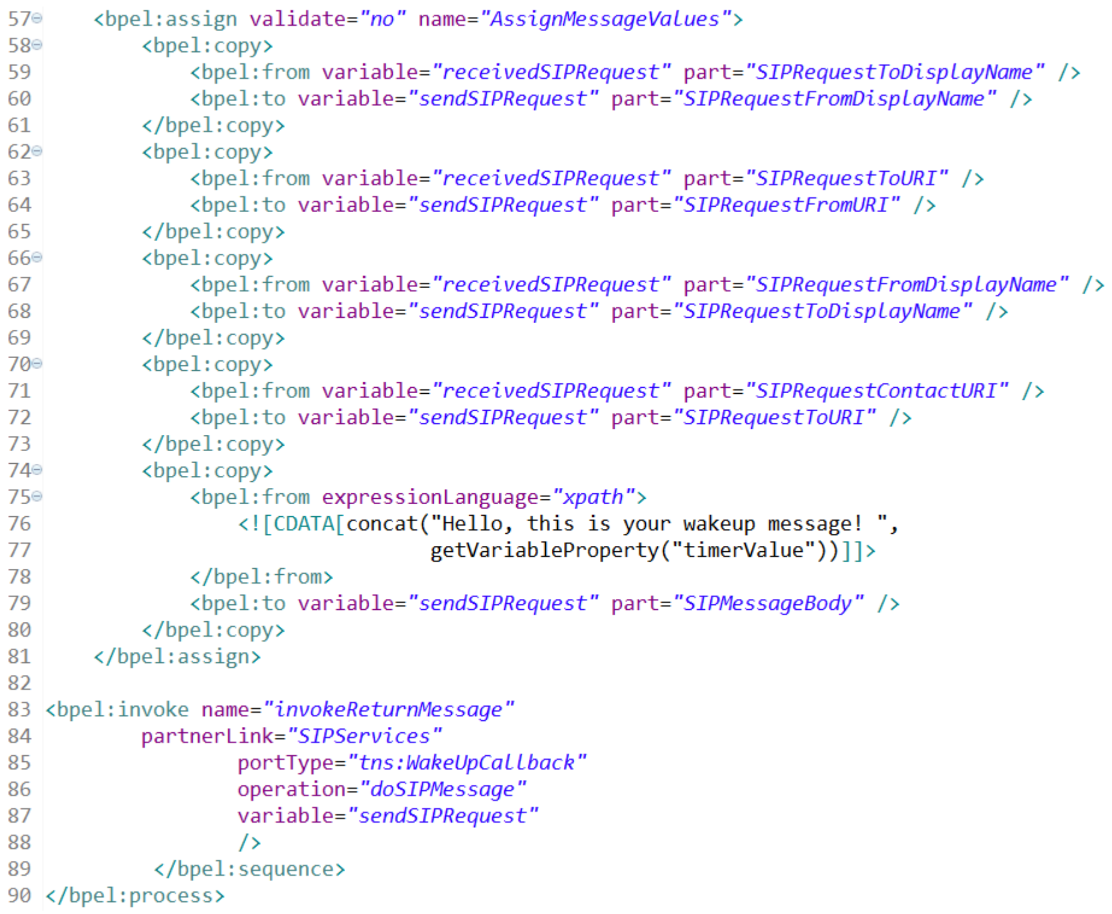
The next activity after the Receive activity is an Assign activity called "AssignTimerValues" (Figure 8.80, line 46 to 51). The contained "`copy`" operation copies the timer value from the "receivedSIPRequest" variable to the "timerValue" variable. With this "timerValue", the Wait activity is initiated (Figure 8.80, line 53 to 55). In the execution phase, the Wait activity will wait until the timer has expired.



```
27 <bpel:sequence name="main">
28     <bpel:assign validate="no" name="AssignInitialValues">
29         <bpel:copy>
30             <bpel:from>
31                 <literal><![CDATA[sip:wakeup@192.168.67.49]]></literal>
32             </bpel:from>
33             <bpel:to variable="receivedSIPRequest" part="SIPRequestToURI"/>
34         </bpel:copy>
35         <bpel:copy>
36             <bpel:from><literal><![CDATA[0]]></literal></bpel:from>
37             <bpel:to variable="timerValue" part="parseInt"/>
38         </bpel:copy>
39     </bpel:assign>
40
41     <bpel:receive name="ReceiveSIPMessage" partnerLink="SIPServices"
42             portType="tns:WakeUpReceive"
43             operation="onSIPMessage" variable="receivedSIPRequest"
44             createInstance="yes"/>
45
46     <bpel:assign validate="no" name="AssignTimerValues">
47         <bpel:copy>
48             <bpel:from variable="receivedSIPRequest" part="SIPMessageBody" />
49             <bpel:to variable="timerValue" part="parseInt" />
50         </bpel:copy>
51         </bpel:assign>
52
53     <bpel:wait name="WaitForTimer" expressionLanguage="xpath">
54         <bpel:until><![CDATA[getVariableProperty("timerValue")]]></bpel:until>
55     </bpel:wait>
```

Figure 8.80: XML document excerpt of the “Wake-up” process – part 2

The third Assign activity “AssignMessageValues” consists of five “copy” operations (Figure 8.81, line 57 to 81). This Assign activity is required to configure the new SIP MESSAGE request “sendSIPRequest”, which is sent as Wake-up message back to the user of the service. The first “copy” operation reads the SIP request “To display name” from the received SIP request and copies this value to the SIP request “From display name” of the “sendSIPRequest”. The next “copy” operation sets the “From URI” of the “sendSIPRequest” by means of the “To URI” of the received SIP request. The third “copy” operation sets the “To display name” of the “sendSIPRequest” by using the “From display name” of the received SIP request. The fourth “copy” extracts the “Contact URI” from the received request and copies it to the “To URI” of the “sendSIPRequest”. The last “copy” operation defines a Wake-up string for the “sendSIPRequest” and adds the defined timer value to the string.



```
57      <bpel:assign validate="no" name="AssignMessageValues">
58          <bpel:copy>
59              <bpel:from variable="receivedSIPRequest" part="SIPRequestToDisplayName" />
60              <bpel:to variable="sendSIPRequest" part="SIPRequestFromDisplayName" />
61          </bpel:copy>
62          <bpel:copy>
63              <bpel:from variable="receivedSIPRequest" part="SIPRequestToURI" />
64              <bpel:to variable="sendSIPRequest" part="SIPRequestFromURI" />
65          </bpel:copy>
66          <bpel:copy>
67              <bpel:from variable="receivedSIPRequest" part="SIPRequestFromDisplayName" />
68              <bpel:to variable="sendSIPRequest" part="SIPRequestToDisplayName" />
69          </bpel:copy>
70          <bpel:copy>
71              <bpel:from variable="receivedSIPRequest" part="SIPRequestContactURI" />
72              <bpel:to variable="sendSIPRequest" part="SIPRequestToURI" />
73          </bpel:copy>
74          <bpel:copy>
75              <bpel:from expressionLanguage="xpath">
76                  <![CDATA[concat("Hello, this is your wakeup message! ",
77                                  getVariableProperty("timerValue"))]]>
78              </bpel:from>
79              <bpel:to variable="sendSIPRequest" part="SIPMessageBody" />
80          </bpel:copy>
81      </bpel:assign>
82
83  <bpel:invoke name="invokeReturnMessage"
84          partnerLink="SIPServices"
85                  portType="tns:WakeUpCallback"
86                  operation="doSIPMessage"
87                  variable="sendSIPRequest"
88                  />
89           </bpel:sequence>
90  </bpel:process>
```

Figure 8.81: XML document excerpt of the “Wake-up” process – part 3

The Invoke activity “invokeReturnMessage” is the last activity of the service description (Figure 8.81, line 83 to 89). It calls the “`doSIPMessage`” operation from the “SIPServices” partner link and takes as variable the previously configured “sendSIPRequest”.

8.4.3 Composition Phase of the Wake-up Service

The service description discussed in the last section is uploaded using the service management servlet and transferred to the management EJB. From there it is handed over to the framework management and stored in the service repository. The service can be triggered for composition and execution by the interactive web interface. The log output of the SEE is shown in Figure 8.82 and Figure 8.83. The correspondent MSC is given in Figure 8.84.

The composition phase is initiated with a configuration event from the framework management. The SCMSBB parses the service description and analyses the "SIPServices" partner link, the three variables and the activities within the service description (Figure 8.82). For all activities found within the service description, the SCMSBB sends configuration events to create and configure the correspondent LSBBs and the required RCSBBs (Figure 8.83).

```
[FrameworkManagementSBB] fire ConfigurationEvent >>
[ServiceControlMSBB] << ConfigurationEvent received
[ServiceControlMSBB] service name:WakeUp
[ServiceControlMSBB] create new service instance -> parsing description
[ServiceControlMSBB] found imports
[ServiceControlMSBB] partnerLinks:
[ServiceControlMSBB] partnerLink: -> name: SIPServices
[ServiceControlMSBB] variables:
[ServiceControlMSBB] Variable: Integer TypeClass: java.lang.Integer
[ServiceControlMSBB] Variable: sipRequestType TypeClass: de.rcsbb.sip.SipRequestType
[ServiceControlMSBB] Variable: sipRequestType TypeClass: de.rcsbb.sip.SipRequestType
[ServiceControlMSBB] found element -> sequence -> name: main
[ServiceControlMSBB] ElementIdentifier: sequence#WakeUp_10582079486864#20
[ServiceControlMSBB] fire ConfigurationEvent >> sequence#WakeUp_10582079486864#20 >>
[ServiceControlMSBB] found element -> assign -> name: AssignInitialValues
[ServiceControlMSBB] ElementIdentifier: assign#WakeUp_10582079486864#22
[ServiceControlMSBB] fire ConfigurationEvent >> assign#WakeUp_10582079486864#22 >>
[ServiceControlMSBB] found element -> receive -> name: ReceiveSIPMessage
[ServiceControlMSBB] ElementIdentifier: receive#WakeUp_10582079486864#24
[ServiceControlMSBB] found element -> rcsbb -> name: SIPServices#WakeUp_10582079486864#26
[ServiceControlMSBB] fire ConfigurationEvent >> SIPServices#WakeUp_10582079486864#26 >>
[ServiceControlMSBB] fire ConfigurationEvent >> receive#WakeUp_10582079486864#24 >>
[ServiceControlMSBB] found element -> assign -> name: AssignTimerValues
[ServiceControlMSBB] ElementIdentifier: assign#WakeUp_10582079486864#28
[ServiceControlMSBB] fire ConfigurationEvent >> assign#WakeUp_10582079486864#28 >>
[ServiceControlMSBB] found element -> wait -> name: WaitForTimer
[ServiceControlMSBB] ElementIdentifier: wait#WakeUp_10582079486864#30
[ServiceControlMSBB] fire ConfigurationEvent >> wait#WakeUp_10582079486864#30 >>
[ServiceControlMSBB] found element -> assign -> name: AssignMessageValues
[ServiceControlMSBB] ElementIdentifier: assign#WakeUp_10582079486864#32
[ServiceControlMSBB] fire ConfigurationEvent >> assign#WakeUp_10582079486864#32 >>
[ServiceControlMSBB] found element -> invoke -> name: invokeReturnMessage
[ServiceControlMSBB] ElementIdentifier: invoke#WakeUp_10582079486864#34
[ServiceControlMSBB] fire ConfigurationEvent >> invoke#WakeUp_10582079486864#34 >>
[ServiceControlMSBB] found element -> rcsbb -> name: SIPServices
[ServiceControlMSBB] ElementIdentifier: SIPServices#WakeUp_10582079486864#36
[ServiceControlMSBB] fire ConfigurationEvent >> SIPServices#WakeUp_10582079486864#36 >>
```

Figure 8.82: Wake-up composition evaluation log – part 1

The SCMSBB waits until it has received the ready events from all components. Then it sends a ready event to the framework management. Upon the framework management has received this event, the service instance is created and ready for execution.

```
[Sequence] << ConfigurationEvent received sequence#WakeUp_10582079486864#20
[Sequence] fire ReadyEvent >> sequence#WakeUp_10582079486864#20 >>
[ServiceControlMSBB] << ReadyEvent received! << sequence#WakeUp_10582079486864#20
[ServiceControlMSBB] Waiting for elements to become ready!
[Assign] << ConfigurationEvent received assign#WakeUp_10582079486864#22
[Assign] fireReadyEvent >> assign#WakeUp_10582079486864#22 >>
[ServiceControlMSBB] << ReadyEvent received! << assign#WakeUp_10582079486864#22
[ServiceControlMSBB] Waiting for elements to become ready!
[SIP_RCSBB] << ConfigurationEvent received << SIPServices#WakeUp_10582079486864#26
[SIP_RCSBB] fire ReadyEvent >> SIPServices#WakeUp_10582079486864#26
[ServiceControlMSBB] << ReadyEvent received! << SIPServices#WakeUp_10582079486864#26
[ServiceControlMSBB] Waiting for elements to become ready!
[Receive] << ConfigurationEvent received receive#WakeUp_10582079486864#24
[Receive] fire ReadyEvent >> receive#WakeUp_10582079486864#24 >>
[ServiceControlMSBB] << ReadyEvent received! << receive#WakeUp_10582079486864#24
[ServiceControlMSBB] Waiting for elements to become ready!
[Assign] << ConfigurationEvent received assign#WakeUp_10582079486864#28
[Assign] fireReadyEvent >> assign#WakeUp_10582079486864#28 >>
[ServiceControlMSBB] << ReadyEvent received! << assign#WakeUp_10582079486864#28
[ServiceControlMSBB] Waiting for elements to become ready!
[Wait] << ConfigurationEvent received wait#WakeUp_10582079486864#30
[Wait] fire ReadyEvent >> wait#WakeUp_10582079486864#30 >>
[ServiceControlMSBB] << ReadyEvent received! << wait#WakeUp_10582079486864#30
[ServiceControlMSBB] Waiting for elements to become ready!
[Assign] << ConfigurationEvent received assign#WakeUp_10582079486864#32
[Assign] fireReadyEvent >> assign#WakeUp_10582079486864#32 >>
[ServiceControlMSBB] << ReadyEvent received! << assign#WakeUp_10582079486864#32
[ServiceControlMSBB] Waiting for elements to become ready!
[Invoke] << ConfigurationEvent received << invoke#WakeUp_10582079486864#34
[Invoke] fire ReadyEvent >> invoke#WakeUp_10582079486864#34
[ServiceControlMSBB] << ReadyEvent received! << invoke#WakeUp_10582079486864#34
[ServiceControlMSBB] Waiting for elements to become ready!
[SIP_RCSBB] << ConfigurationEvent received << SIPServices#WakeUp_10582079486864#36
[SIP_RCSBB] fire ReadyEvent >> SIPServices#WakeUp_10582079486864#36
[ServiceControlMSBB] << ReadyEvent received! << SIPServices#WakeUp_10582079486864#36
[ServiceControlMSBB] fire ReadyEvent >> WakeUp_10582079486864
[FrameworkManagementSBB] << ReadyEvent received << WakeUp_10582079486864
[FrameworkManagementSBB] <---> WakeUp_10582079486864 created and configured <--->
```

Figure 8.83: Wake-up composition evaluation log – part 2

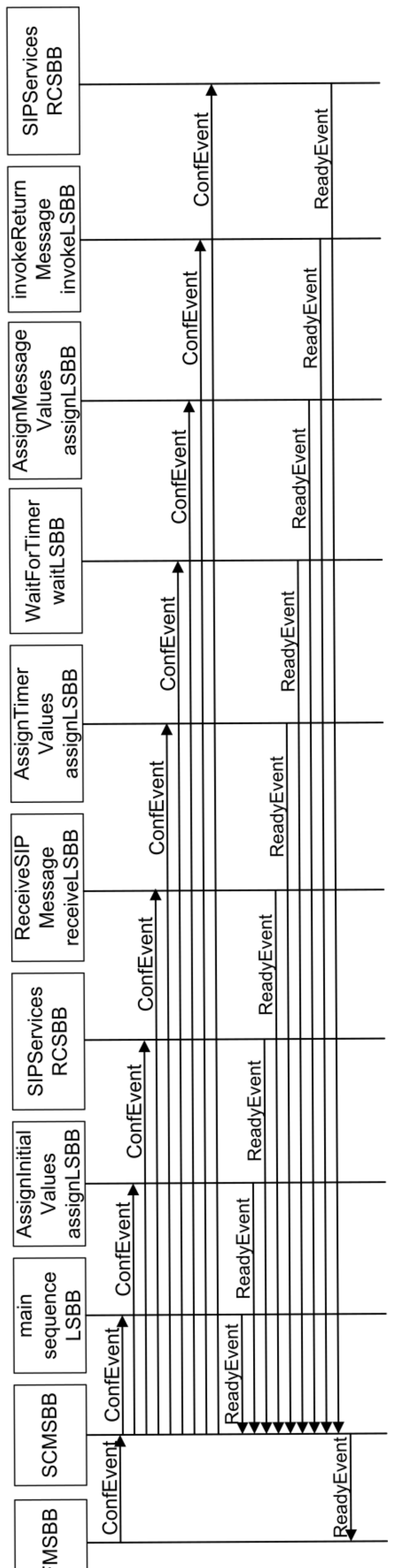

FMSBB
SCMSBB
main sequence LSBB
AssignInitial Values assignLSBB
SIPServices RCSBB
ReceiveSIP Message receiveLSBB
AssignTimer Values assignLSBB
WaitForTimer waitLSBB
AssignMessage Values assignLSBB
invokeReturn Message invokeLSBB
SIPServices RCSBB
ConfEvent
ReadyEvent

Figure 8.84: Wake-up composition phase MSC

8.4.4 Execution Phase of the Wake-up Service

After the service creation phase is completed successfully, the generated service instance is ready to be triggered for execution by the framework management. The log output of the service instance is given in Figure 8.85 to Figure 8.91. The Wake-up message from the service is shown in a screenshot of the SIP softphone "PhonerLite" (Figure 8.92). The MSC of the event communication in the service execution phase is shown in Figure 8.93.

To start the execution, an inter-service event is sent to the SCMSBB (Figure 8.85). The execution of the LSBBs occurs in the order described in the BPEL process. When one component is executed, it sends inter-service events to the following element. The SCMSBB starts sending an event to the Sequence LSBB, and the Sequence LSBB sends an event to the Assign LSBB.

```
[FrameworkManagementSBB] <> triggering WakeUp_10582079486864 for execution <>
[FrameworkManagementSBB] fire InterServiceEvent >>
[ServiceControlMSBB] << InterServiceEvent received << WakeUp_10582079486864
[ServiceControlMSBB] fire InterServiceEvent >> WakeUp_10582079486864 >>
[Sequence] << InterServiceEvent received << sequence#WakeUp_10582079486864#20
[Sequence] fire InterServiceEvent >> sequence#WakeUp_10582079486864#20 >>
[Assign] << InterServiceEvent received assign#WakeUp_10582079486864#22
[Assign]  Copy operation: new TO variable value: 0
[Assign] fire InterServiceEvent >> assign#WakeUp_10582079486864#22 >>
[SIP_RCSBB] << InterServiceEvent received SIPServices#WakeUp_10582079486864#26
[SipResourceAdaptor] Received Request:
```

Figure 8.85: Wake-up execution evaluation log – part 1

The next SBB after the Assign LSBB is the SIP RCSBB. If the SIP RCSBB receives the inter-service event from the Assign LSBB, it starts waiting for an "onMessageRequest" event from the SIP RA. When the SIP RA receives such an event for the Wake-up service, it sends an "onMessageRequest" event to the SIP RCSBB. The RCSBB is triggered, when the service receives a SIP MESSAGE request from the service user. Figure 8.86 displays the received request. This request was sent from the SIP user agent “PhonerLite” which has the “From URI”: sip:testuser@192.168.67.15. The user of the service requests a Wake-up message, which should be sent in 5000 milliseconds back to the user.

```
MESSAGE sip:wakeup@192.168.67.49 SIP/2.0
Via: SIP/2.0/UDP 192.168.67.15:5060;branch=z9hG4bK00e67c29b47ae3119bdd0800 . . .
From: "PhonerLite" <sip:testuser@192.168.67.15>;tag=3227538128
To: <sip:wakeup@192.168.67.49>
Call-ID: 00E67C29-B47A-E311-9BDC-08002700BC69@192.168.67.15
CSeq: 2 MESSAGE
Contact: <sip:testuser@192.168.67.15:5060>
Content-Type: text/plain;charset="UTF-8"
Max-Forwards: 70
Date: Mon, 13 Jan 2014 11:33:16 GMT
User-Agent: SIPPER for PhonerLite
Content-Length: 6

5000
```

Figure 8.86: Wake-up execution evaluation log – part 2

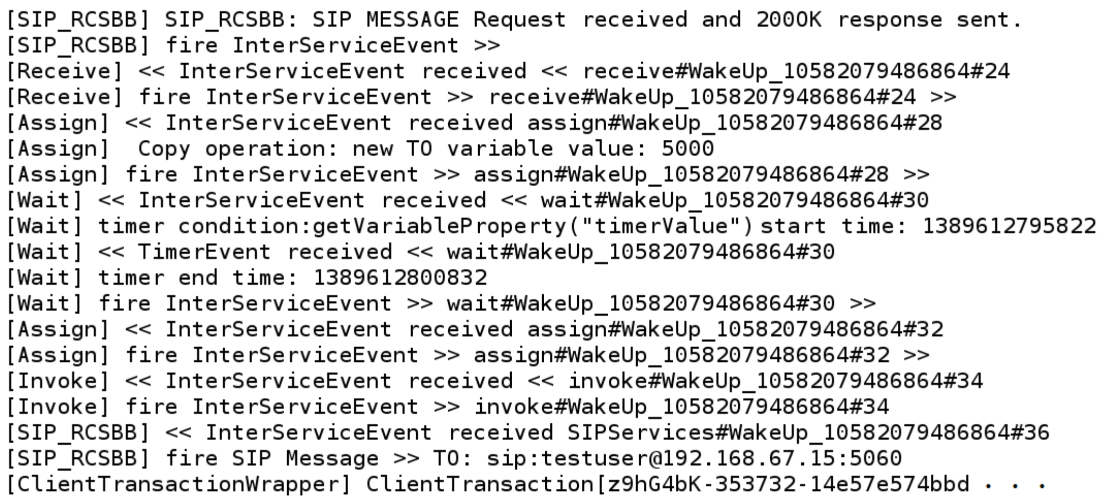
The SIP RCSBB receives the “onMessage” event from the SIP RA and calls the “`sendResponse()`” method on the SIP RA. The RA sends out a 200 OK SIP response to the softphone to end the SIP transaction. This SIP response message is shown in Figure 8.87.

```
[SIP_RCSBB] << SIP MESSAGE received <<
[ServerTransactionWrapper] ServerTransaction[z9hG4bK00e67c2 . . .  sending response
SIP/2.0 200 OK
To: <sip:wakeup@192.168.67.49>
Via: SIP/2.0/UDP 192.168.67.15:5060;branch=z9hG4bK00e67c29b47ae3119 . . .
CSeq: 2 MESSAGE
Call-ID: 00E67C29-B47A-E311-9BDC-08002700BC69@192.168.67.15
From: "PhonerLite" <sip:testuser@192.168.67.15>;tag=3227538128
Content-Length: 0
```

Figure 8.87: Wake-up execution evaluation log – part 3

Upon the SIP RCSBB has been executed, it sends an inter-service event to the next component of the service, the receive LSBB (Figure 8.88). Then the Assign LSBB, and afterwards the Wait LSBB, is executed. In the Wait LSBB, the service instance waits for the defined duration. In this case, it waits 5000 milliseconds, before the next component, the Assign LSBB, is executed. The Assign LSBB is followed by the Invoke LSBB.

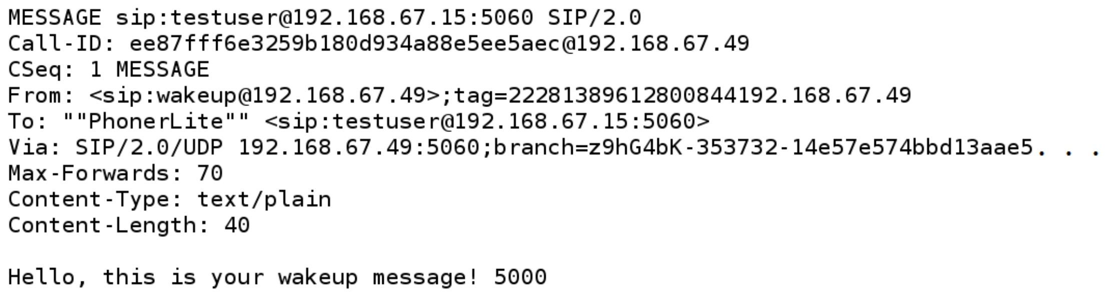
The Invoke LSBB triggers the sending of the Wake-up message back to the user of the service. It sends an inter-service event to the SIP RCSBB, which creates the SIP MESSAGE request and calls the SIP RA to send it to the user of the service.



```
[SIP_RCSBB] SIP_RCSBB: SIP MESSAGE Request received and 200OK response sent.
[SIP_RCSBB] fire InterServiceEvent >>
[Receive] << InterServiceEvent received << receive#WakeUp_10582079486864#24
[Receive] fire InterServiceEvent >> receive#WakeUp_10582079486864#24 >>
[Assign] << InterServiceEvent received assign#WakeUp_10582079486864#28
[Assign]  Copy operation: new TO variable value: 5000
[Assign] fire InterServiceEvent >> assign#WakeUp_10582079486864#28 >>
[Wait] << InterServiceEvent received << wait#WakeUp_10582079486864#30
[Wait] timer condition:getVariableProperty("timerValue") start time: 1389612795822
[Wait] << TimerEvent received << wait#WakeUp_10582079486864#30
[Wait] timer end time: 1389612800832
[Wait] fire InterServiceEvent >> wait#WakeUp_10582079486864#30 >>
[Assign] << InterServiceEvent received assign#WakeUp_10582079486864#32
[Assign] fire InterServiceEvent >> assign#WakeUp_10582079486864#32 >>
[Invoke] << InterServiceEvent received << invoke#WakeUp_10582079486864#34
[Invoke] fire InterServiceEvent >> invoke#WakeUp_10582079486864#34
[SIP_RCSBB] << InterServiceEvent received SIPServices#WakeUp_10582079486864#36
[SIP_RCSBB] fire SIP Message >> TO: sip:testuser@192.168.67.15:5060
[ClientTransactionWrapper] ClientTransaction[z9hG4bK-353732-14e57e574bbd . . .
```

Figure 8.88: Wake-up execution evaluation log – part 4

The SIP MESSAGE request with the Wake-up message is shown in Figure 8.89. It is sent to the request URI: "sip:testuser@192.168.76.15". The Wake-up message includes the text, "Hello, this is your wakeup message! 5000".



```
MESSAGE sip:testuser@192.168.67.15:5060 SIP/2.0
Call-ID: ee87fff6e3259b180d934a88e5ee5aec@192.168.67.49
CSeq: 1 MESSAGE
From: <sip:wakeup@192.168.67.49>;tag=22281389612800844192.168.67.49
To: ""PhonerLite"" <sip:testuser@192.168.67.15:5060>
Via: SIP/2.0/UDP 192.168.67.49:5060;branch=z9hG4bK-353732-14e57e574bbd13aae5. . .
Max-Forwards: 70
Content-Type: text/plain
Content-Length: 40

Hello, this is your wakeup message! 5000
```

Figure 8.89: Wake-up execution evaluation log – part 5

The SIP RCSBB is the last component of the service description. After having sent out the SIP MESSAGE request, the SIP RCSBB is completed and sends an inter-service event to the SCMSBB. The service instance has completed its execution, and the SCMSBB sends an inter-service event as confirmation to the framework management.

```
[SIP_RCSBB] fire InterServiceEvent >> SIPServices#WakeUp_10582079486864#36
[ServiceControlMSBB] << InterServiceEvent received << WakeUp_10582079486864
[ServiceControlMSBB] fire InterServiceEvent >> WakeUp_10582079486864 >>
[FrameworkManagementSBB] << InterServiceEvent received
[FrameworkManagementSBB]  -> WakeUp_10582079486864-> executed
[SipResourceAdaptor] Received Response:
```

Figure 8.90: Wake-up execution evaluation log – part 6

The SIP MESSAGE request is sent to the “PhonerLite” softphone. The softphone answers with a SIP 200 OK response to complete the SIP transaction (Figure 8.91), and the received Wake-up message is displayed in the “PhonerLite” message window (Figure 8.92).

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP 192.168.67.49:5060;branch=z9hG4bK-353732-14e57e574bbd13aa . . .
From: <sip:wakeup@192.168.67.49>;tag=22281389612800844192.168.67.49
To: "" <sip:testuser@192.168.67.15:5060>;tag=80d6772cb47ae3119bdd08002700bc69
Call-ID: ee87fff6e3259b180d934a88e5ee5aec@192.168.67.49
CSeq: 1 MESSAGE
Contact: <sip:testuser@192.168.67.15:5060>
Allow: INVITE,OPTIONS,ACK,BYE,CANCEL,INFO,NOTIFY,MESSAGE,UPDATE
Server: SIPPER for PhonerLite
Date: Mon, 13 Jan 2014 11:33:21 GMT
Content-Length: 0
```

Figure 8.91: Wake-up execution evaluation log – part 7

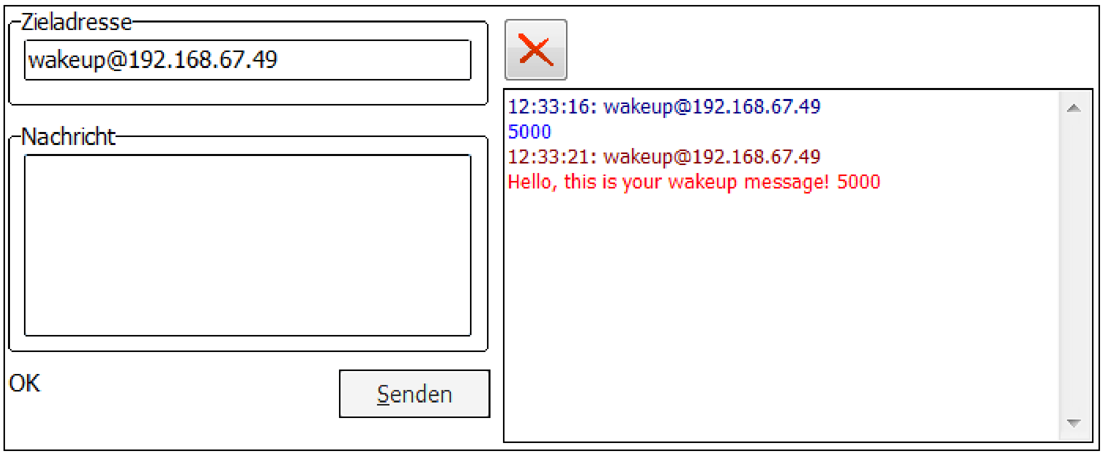

Zieladresse
wakeup@192.168.67.49
Nachricht
OK
12:33:16: wakeup@192.168.67.49
5000
12:33:21: wakeup@192.168.67.49
Hello, this is your wakeup message! 5000

Figure 8.92: Wake-up message in SIP user agent

The complete MSC of the service execution phase is shown in Figure 8.93.

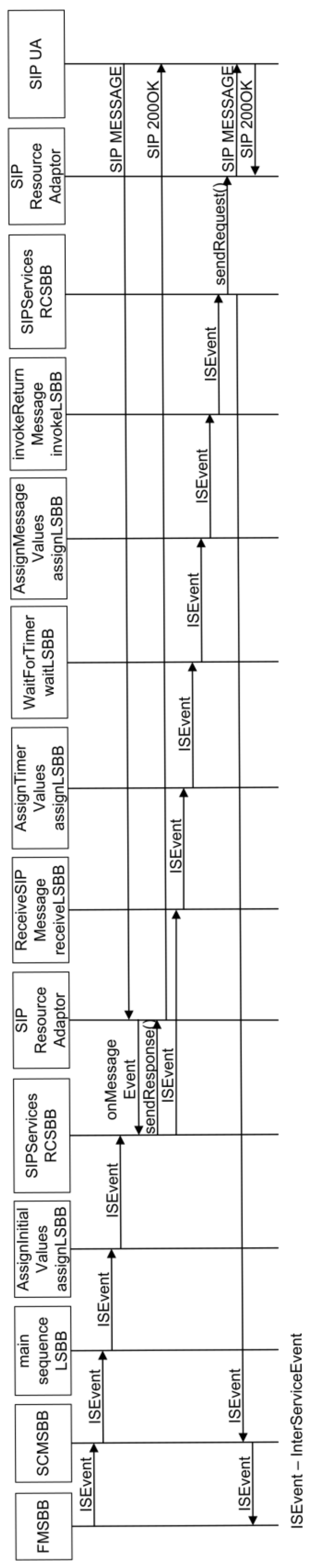


Figure 8.93: Wake-up execution phase MSC

8.5 Analyses of the Quantitative Requirements of the Framework Prototype

This section analyses the framework prototype. For this purpose, one example service will be defined. This service will be developed in two versions. The first version is developed as conventional value-added service with JSLEE and the second version with the framework developed in this PhD thesis. Both services are compared with each other; in this context their length of the code, their development time, their numbers of SBBs, their latency, and their throughput are analysed. Finally, based on the example service, the scalability of services within the developed framework is analysed.

8.5.1 Evaluation Scenario

The service that is used for the evaluation is a Chat service that is based on the SIP protocol. The service uses SIP messages to transfer the chat data. Multiple participants can log in into these chat services. They can use their SIP user agents to communicate with the service. The user can log in into the service and logout from the service. The chat data that is received by one user is sent to all other users in the chat room. Multiple chat rooms can be realised by starting multiple service instances with different room names.

The conventionally developed service consists of one SBB. It was developed with the Java programming language. The service that was developed with the proposed framework consists of 20 SBBs. It was described with the Eclipse BPEL developer tool. The graphical BPEL representation of the service is shown in Figure 8.94.

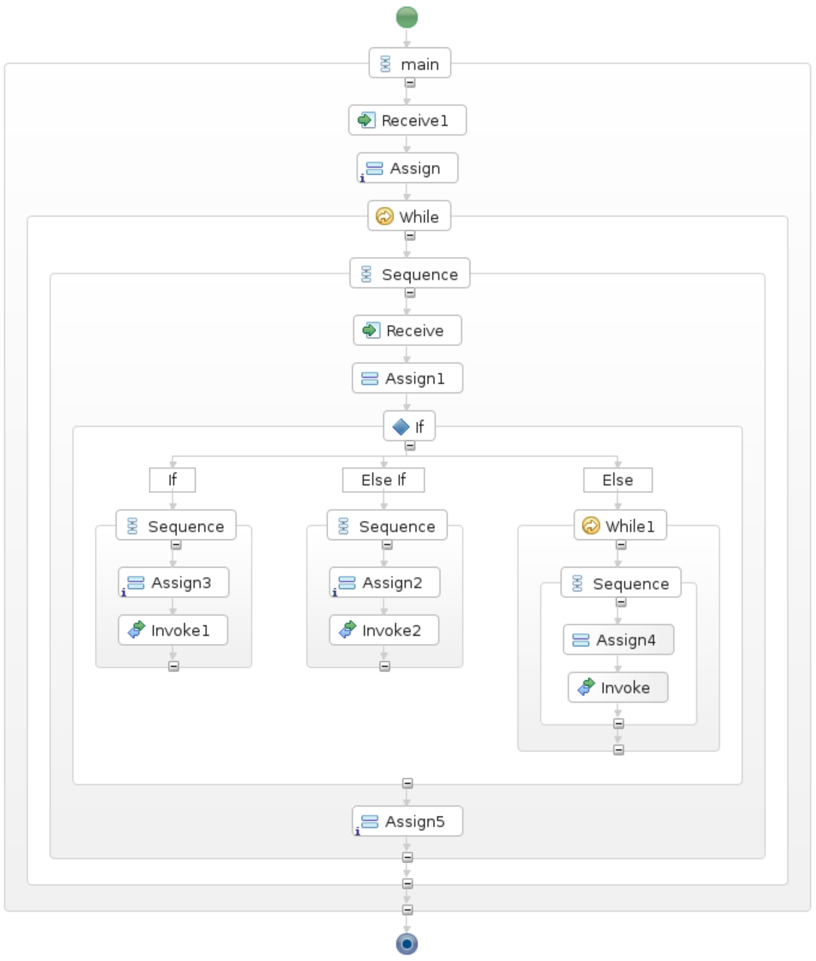


Figure 8.94: Graphical BPEL representation of the chat service

Within the main sequence, the process begins with the initialisation of the service; the assign activity “Assign” configures the SIP URI of the chat room and initialises some variables.

The while loop “While” contains the main logic of the service. Within this loop, the chat room functionality is described. In the service execution phase, this loop runs until the service is stopped by the framework management.

The receive activity “Receive” listens for a SIP MESSAGE request that is sent to this chatroom. The required information from the received SIP MESSAGE is stored in variables by the assign activity “Assign1”.

The content of the SIP MESSAGE is evaluated in the “if” activity. The left if-branch is taken when the SIP MESSAGE contains the content “login”. In this case, the sender of the SIP MESSAGE tries to log in into the service and is added to the chat user list. A confirmation is sent back to this user. The second if-branch in the middle of the “if”-activity shown in Figure 8.94 is executed when the content of the SIP MESSAGE contains “logout”. In this case, the correspondent chat user is removed from the user list, and a confirmation is sent to this user. In case that the SIP MESSAGE content contains a chat message, the right if-branch is executed. This branch contains another while-loop, “While1”. Within this if-branch, the received text message is sent to all connected chat users.

After one of the if-branches has been executed, some variables are configured in the assign activity “Assign5”, and the while-loop starts again. The service waits for the next SIP message.

Both of the services, the manually developed service and the service which is generated from the BPEL service description, offer the same functionality. The protocol-specific communication for both services consists of three parts, the login, the chat communication, and the logout.

The MSC in Figure 8.95 represents the login process of user agent A (UA A) into the service.

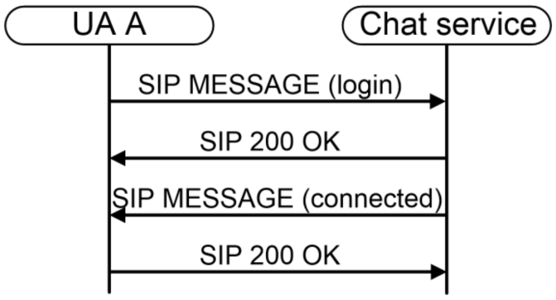


Figure 8.95: Chat service login

The login process is initiated with a SIP MESSAGE request sent to the Chat service. It contains the content “login”. This SIP request is answered with a 200 OK SIP response. The chat service creates a SIP MESSAGE request which contains a confirmation message. The SIP MESSAGE is sent to UA A, to inform that the login process was successful. Now, UA A is connected to the chat service.

The chat functionality itself is depicted in Figure 8.96. There, UA A is sending a SIP MESSAGE request with the content “hello”. The server answers the SIP request with a 200 OK response. In the next step, the Chat service sends the text of the received SIP MESSAGE request to all other participants that are registered to the service. The UAs confirm the reception of the SIP request with a SIP 200 OK response.

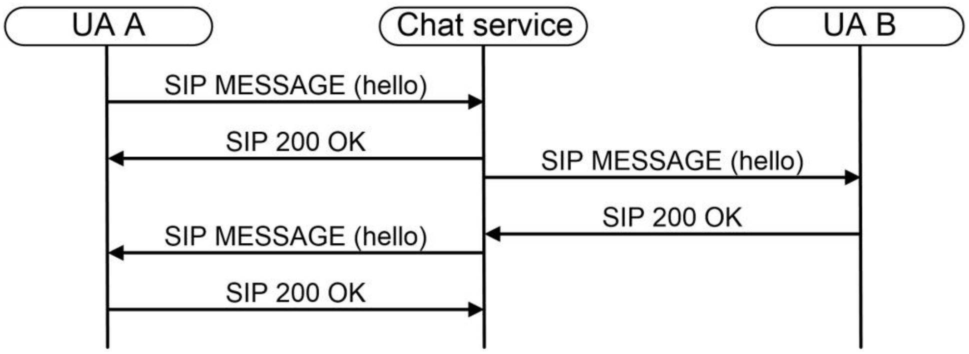


Figure 8.96: Chat service text exchange

The logout process (Figure 8.97) is triggered by the UA by sending a SIP MESSAGE request with the content “logout” to the service. The service answers the request with a 200 OK response. The successful logout is confirmed by the service with a SIP MESSAGE request that contains the disconnection information. This request is answered with a SIP 200 OK response by the SIP UA.

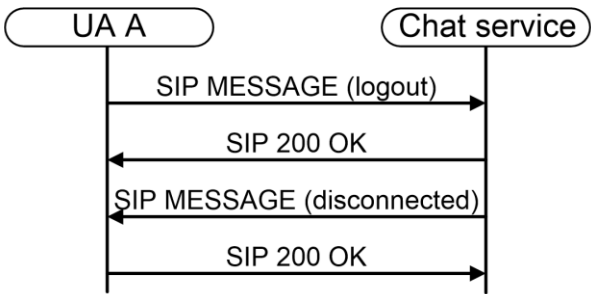


Figure 8.97: Chat service logout

8.5.2 Comparison between Conventional and PhD Prototype Service Creation

The service that was described in the evaluation scenario in section 8.5.1 was developed for the JSLEE framework and for the framework proposed in this thesis. The manually developed JSLEE service was created with the Eclipse Java IDE and the service for the proposed framework with the Eclipse BPEL developer. A comparison between the two approaches is given in Table 8.1.

Table 8.1: Comparison between the approaches

	Conventionally developed JSLEE value-added service	Value-added service generated with the proposed framework
Development time	24 h	5 h
Lines of code	Java file: 252 lines	BPEL file: 196 lines

	sbb-jar file: 59 lines deployable unit: 8 lines service.xml: 19 lines build.xml: 109 lines total: 447	total: 196 lines
Number of SBBs	1 SBB	20 SBBs

The development of the JSLEE service requires three working days (24h), and the service has a length of 447 lines of code. The service that was developed with the proposed framework requires only 5 hours development time. The service was developed with the Eclipse BPEL developer in a graphical way, and the length of the XML code is only 196 lines. The manual development of the Chat service requires more than twice as much lines of code and lasts nearly 5 times longer than the service created with the proposed framework. Furthermore, the manually developed service requires knowledge about Java programming, SIP, and, additionally, knowledge about writing the support files like deployment files. For describing the service with the proposed framework, only basic BPEL knowledge is required. Therefore, the development of this service with BPEL is faster and simpler than the development of the conventional JSLEE service.

In general, the BPEL code is shorter than the Java code. The service description in BPEL is more abstract; it is not required to care about all the details like in Java. The service for the proposed framework requires no descriptor and deployment files. Apart from that, the graphical service description speeds up the development process.

8.5.3 Performance Analysis of the Framework Prototype

In this section, the performance of services that are developed with the proposed framework is analysed. The performance of a service that is created for the proposed framework is measured and compared with an equivalent service created conventionally for the JAIN-SLEE framework. For the performance analysis, the Chat service scenario from section 8.5.1 is used again.

To compare the performance of services that consist of one SBB with services that consist of more SBBs, one SBB was used for the manually developed service and 20 SBBs for the service that was generated with the proposed framework.

It is expected that the approach which consists of multiple SBBs will lead to worse latency compared to the approach that consists of one SBB.

Relevant for the performance of telecommunication services is the throughput and the latency. The latency is the time that is required for transmitting a signal from the sender to the receiver. In this scenario, instead of the latency, the round trip time (RTT) is measured. The RTT is the time from sending a signal until receiving the answer for this signal. In this scenario, it is the duration of a timer that starts when a SIP MESSAGE request is sent out, and stops when the response for this SIP MESSAGE is received. The SIPp testing tool simulates the required SIP user agents and measures the throughput and the latency.

Both versions of the service are tested on the same machine with 3 GB RAM and 2 * 1.7 MHz processors. A Debian 6 operation system is installed on this computer with a 2.6.32-5-686 Linux Kernel. The SIPp test script is running on a separate computer.

As JSLEE implementation, Mobicents 2.4.1 final (Mobicents, 2014) is deployed on a JBOSS v5.1.0 application server.

For testing the throughput of the service, the SIPp tool is configured to send out SIP messages to the service. The logout scenario is used again in order to measure the throughput. The SIP communication is shown in Figure 8.98.

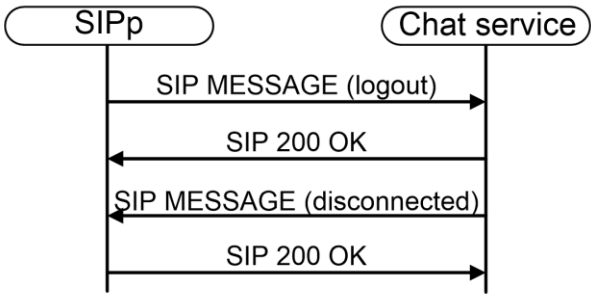


Figure 8.98: Chat service logout scenario

In the service scenario with one SBB, all service logic is implemented in this SBB (Figure 8.99).

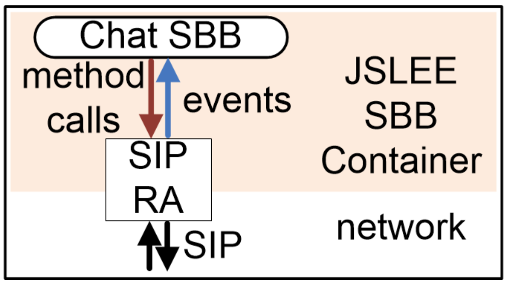


Figure 8.99: Chat service with one SBB

To receive and to generate SIP protocol messages, the service communicates with the SIP RA. It is listening for events from the SIP RA and calling methods on the SIP RA to generate and send SIP responses and SIP MESSAGE requests.

The service that is generated with the PhD Framework consists of 20 SBBs. These SBBs need to be created and configured before the service can be executed.

In the execution phase, the framework management starts the execution of the service. In the BPEL process (Figure 8.94), the outer "While" loop is executed until it reaches the "receiveLSBB" (Figure 8.100) within this loop. The "receiveLSBB" waits for an incoming SIP MESSAGE request from a user agent. The "whileLSBB" and its containing SBBs are executed each time a SIP MESSAGE request is received (Figure 8.100).

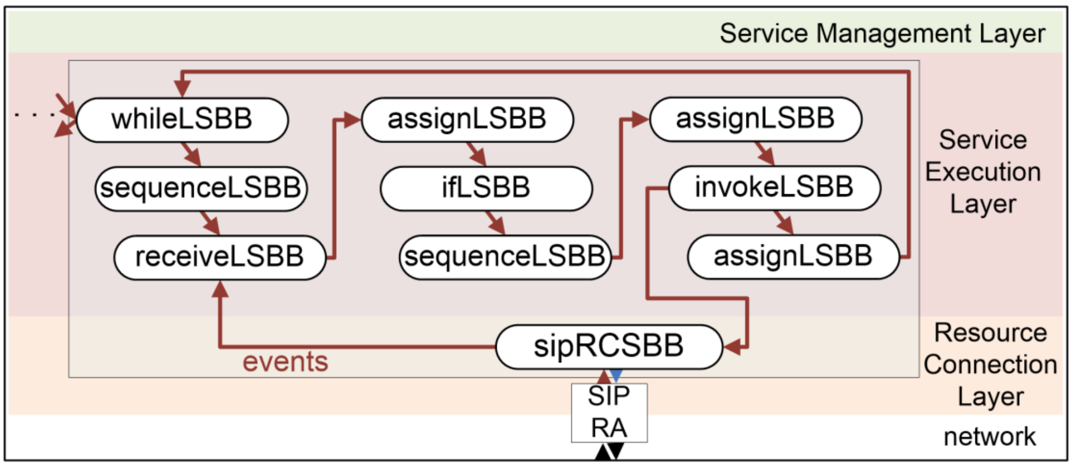


Figure 8.100: Chat service with multiple SBBs

For each time the logout scenario is triggered, 4 SIP messages must be handled by the service; the incoming SIP MESSAGE request, the corresponding 200 OK response, the SIP MESSAGE request with the login confirmation, and the 200 OK response for this confirmation message.

The SIPp tool controls the SIP messages that are sent to the service and measures the number of successful logouts in calls per second (CPS) that can be handled by the service. This is done by increasing the number of logouts until messages are lost or not handled correctly. The results of these tests are presented in the diagram in Figure 8.101.

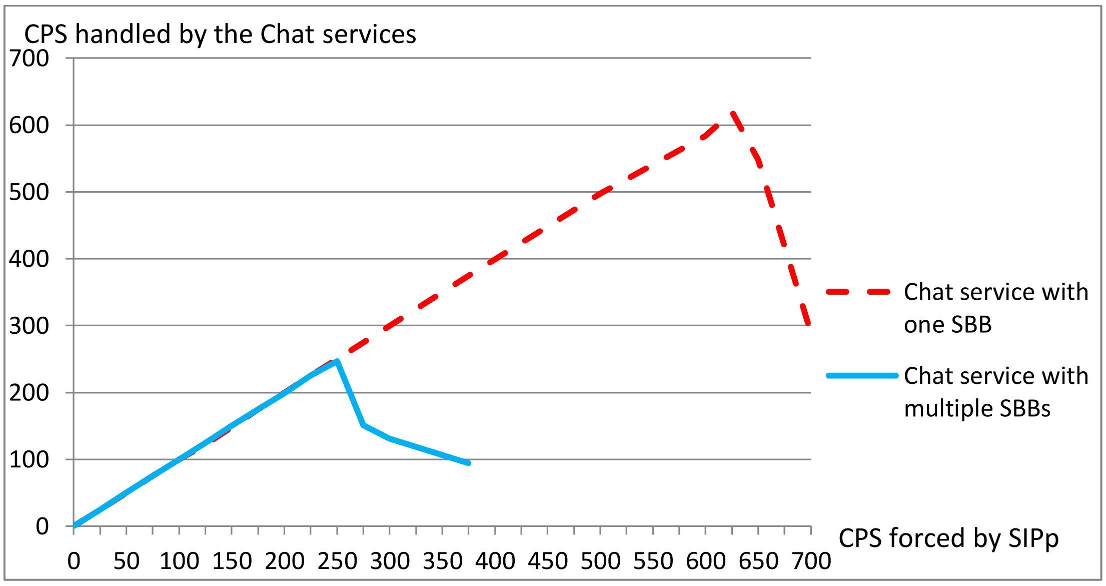


Figure 8.101: Throughput test results

The SIPp tool defines a SIP message communication with the same call ID as call, which is why the performance of the PhD prototype is measured in CPS. On the X-axis, the logouts in CPS are plotted that are forced by the SIPp testing tool. The Y-axis represents the logouts in CPS that are achieved by the services. The red dotted line represents the successful logouts per second with one SBB, and the blue line represents the successful logouts per second with multiple SBBs.

As expected, the manually developed service with only one SBB is able to handle more logouts per second than the service which consists of multiple SBBs. The multiple SBB service reaches a maximum of 252.1 CPS, and the service which consists of one SBB reaches 619.5 CPS. Both of the services reach good results, also the service that is generated from 20 SBBs reaches more than 250 CPS.

The results can be explained with the more complex internal communication between the SBBs in the multiple SBB scenarios and with the developed PhD framework prototype implementation, which is not optimised for performance testing. For simple services, the single SBB approach is more is more efficient than a solution that consists of multiple SBBs, but complex services cannot be mapped to one SBB only. Furthermore, JSLEE services require multiple SBBs, e.g. for parallel execution.

The next performance criterion is the round trip time (RTT). Again, the logout scenario is selected for this test (Figure 8.102).

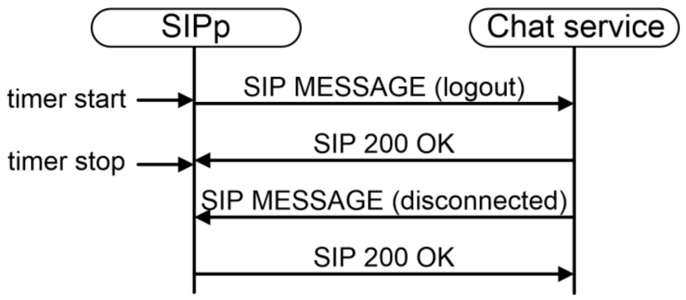


Figure 8.102: Round trip time Timer for the SIPp logout scenario

To measure the RTT, SIPp starts a timer when the SIP MESSAGE request with the logout command is sent to the service. The timer is stopped when the 200OK response is received. The RTT of both services is measured in dependence with the number of logouts per second in CPS. The CPS value is controlled by SIPp. The RTT is measured in milliseconds of some specific CPS values; each CPS value is measured for a period of 5000 calls. The results of these tests are displayed in Figure 8.103.

The diagram shows the average RTT in ms of both Chat services. The CPS value is represented by the X-axis and the RTT by the Y-axis. The values for the single SBB Chat service are represented with a red dotted line, and the values of the service with multiple SBBs are represented as a blue line.

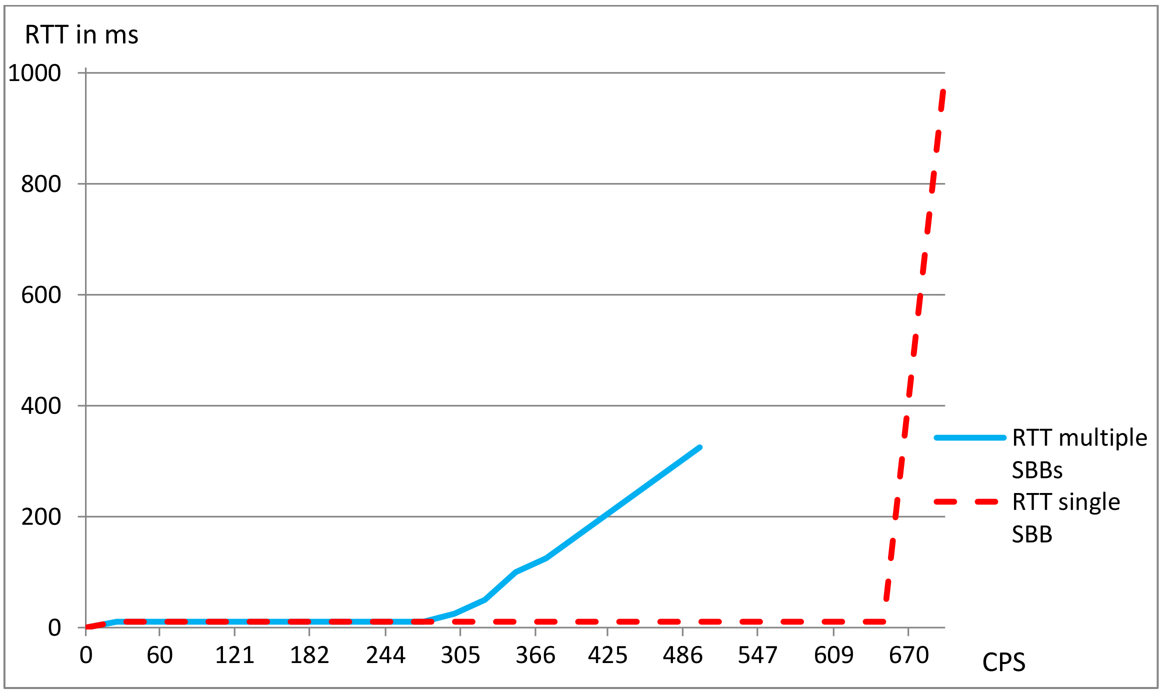


Figure 8.103: Chat service round trip time (RTT)

Both services reach an average RTT of 10ms. This value is constant until they reach their maximum CPS value. The multiple SBB Chat service supports a maximum of 252.1 CPS. Higher CPS values influence the average RTT negatively, the service is not able to handle all SIP requests in time, and SIP responses are delayed or dropped. The service that consists of one SBB supports 619.5 CPS. Increasing this max CPS value with SIPp will lead to delayed or dropped SIP responses.

The test system is not able to handle all logouts correctly beyond the maximum CPS. This means that the computing resources of the test computer might be the bottleneck. Figure 8.104 shows the dependence between CPS and CPU load.

The diagram below shows the number of logouts in CPS triggered by SIPp on the X-axis and the CPU load on the Y-axis. The results of the single SBB service are represented as a red dotted line and the results of the multiple SBB service as a blue line.

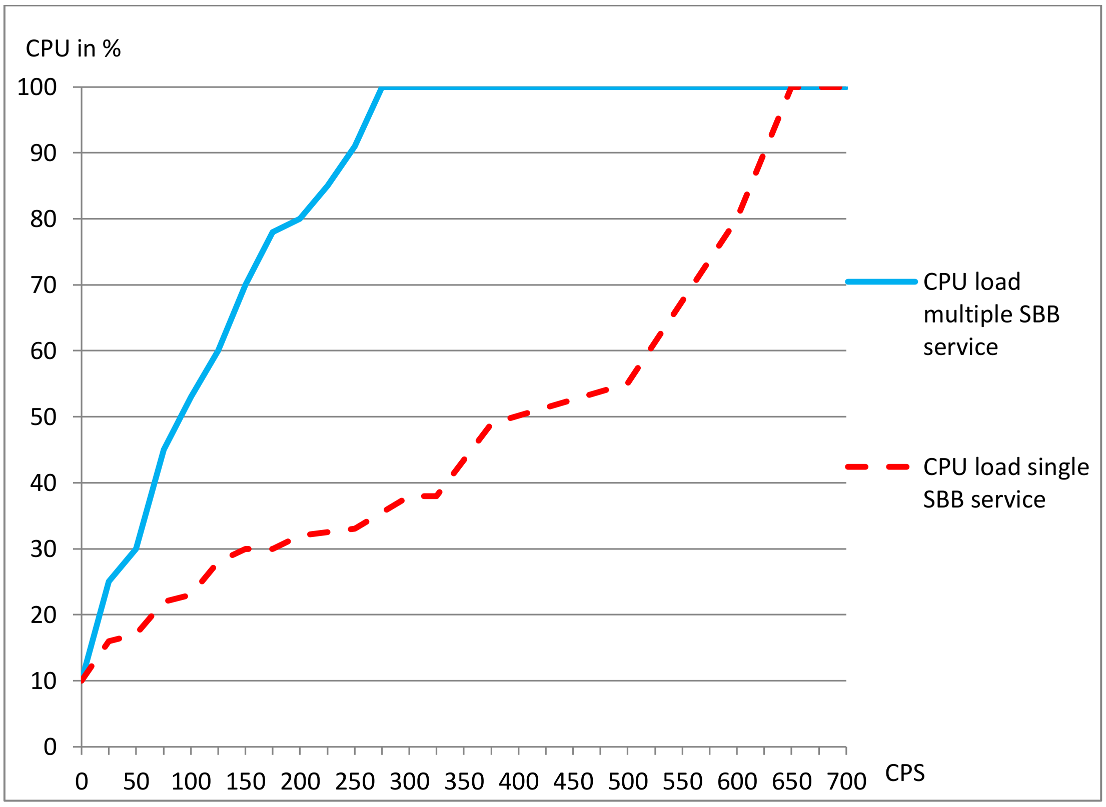


Figure 8.104: Chat service CPU load

As expected, both services reaching around 100% CPU load at their maximum CPS values. This means that the processor resources are the bottleneck for the maximum CPS. The average CPU load without running a service is about 9 MHz. The multiple SBB service reaches the 100% CPU load at about 250 CPS, the single SBB solution at about 620 CPS. The result shows that the single SBB solution reaches a higher performance than the solution with 20 SBBs. The event-based communication and the handling of the SBBs require some CPU load, but as already said above, complex JSLEE service cannot, too, consists of only one SBB, and the developed research prototype is not optimised for performance testing. The reached results are very good, and the advantages (refer to section 9.2) of the multiple SBB solution prevail.

8.5.4 Service Scalability

This section contains a discussion on the scalability of multiple service instances. The prototype developed in this PhD thesis is able to handle multiple service instances. This has the advantage that user requests can be handled simultaneously. In this section, the scalability of these service instances is analysed.

To analyse the scalability of the prototype, a test scenario is defined. Here again, the logout scenario with the Chat service from the previous section (refer to section 8.5.3) is used. The number of possible service instances is limited by the system RAM. The test system is a 32-bit system with 3 GB RAM. For the prototype including the Java VM, the JBOSS application server, and the JSLEE Mobicents framework, 1700 MB of RAM was available.

Relevant for the scalability are the logouts per second in CPS in dependence of the number of instances (Figure 8.105).

The figure shows the scalability of the service instances. On the X-axis the number of instances is displayed, and on the Y-axis the logouts per second are shown in the unit of CPS. The blue line represents the maximum reached CPS value for a specific number of instances.

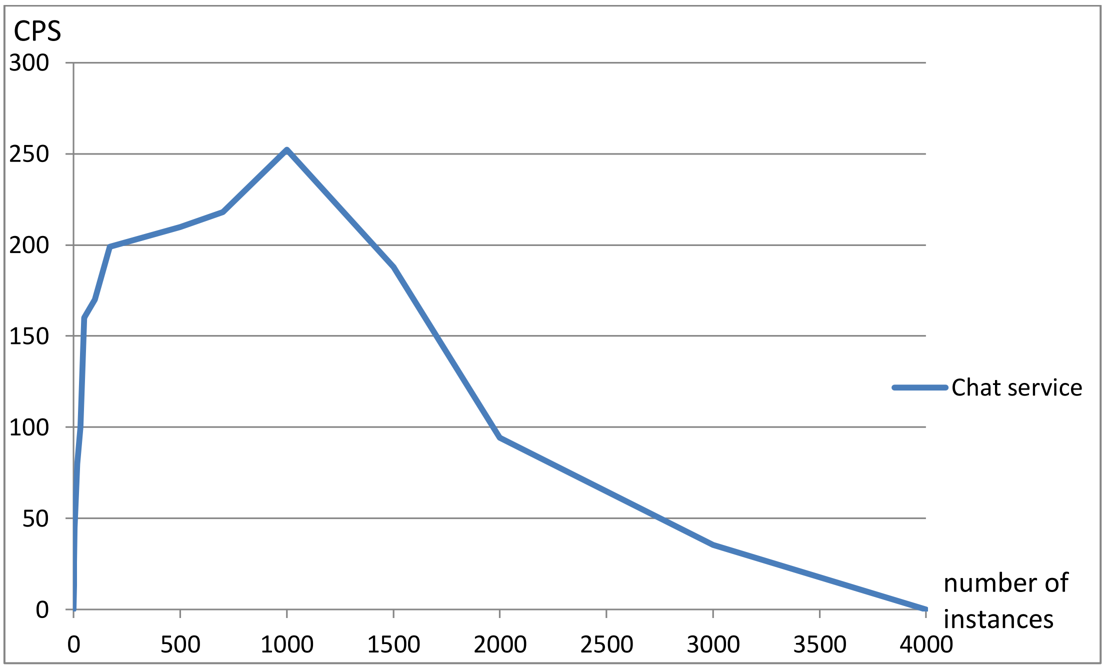


Figure 8.105: Framework scalability

Increasing the number of parallel running instances also increases the maximum supported CPS value, until a number of 1000 service instances are reached. A further increase of the number of instances will cause a drop of logouts per second. With more than 1000 instances, there is not enough free RAM available for a fast execution of the service instances. The JVM needs to handle/swap the memory and requires therefore more CPU resources. The maximum number of instance that can be instantiated with the test machine is around 4000. However, the execution of a service instance also requires RAM. When applying the logout scenario to the 4000 instances, the CPU load rises to 100%, and most SIP messages get lost or the instances stop working. Higher numbers of instances lead to the same behaviour or directly to “Out of Memory” exceptions.

This analysis leads to the result that the number of instances influences the CPS. Rising the number of instances also offers a support of more CPS. However, the number of instances is limited by the available RAM. In addition, the execution of the service instances requires free RAM and CPU.

The developed framework is scalable. The scalability is limited by the available amount off RAM. Therefore, new service instances can be generated and executed until the maximum amount of RAM is used and the Java VM starts swapping the RAM to the hard drive.

8.6 Conclusion

In this chapter, the framework was evaluated regarding the defined requirements. The requirements were analysed whether or not they are fulfilled within the framework. The proposed framework meets all requirements successfully.

In the next step, the architecture of the prototype was systematically introduced. The prototype implements the most important components of the SCE and SEE to allow the creation of an example service for the proof of concept.

The required SEE components, which are involved in the service instance life cycle, were described and evaluated. For all components the whole life cycle, – from the service description and the automatic creation to the execution – was presented and the results were analysed. The research prototype was successfully adopted for a proof of concept evaluation of the proposed framework, which demonstrates its functionalities as well as its general applicability.

The proposed novel concept was demonstrated using a typical value-added service. A Wake-up service was described with a BPEL design tool. A similar scenario exists

(Martins, 2011), which was defined for conventional means of service development with Java for mobicents JAIN SLEE.

In the previous section, the framework prototype was practically analysed. An evaluation scenario was defined, and a value-added service which was created with the framework was compared with a conventional JSLEE service. This conventional scenario requires substantiated knowledge of Java, SIP, and XML. Moreover, it will take three days to develop the value-added service.

The service for the PhD framework was developed with a graphical BPEL tool, no special knowledge about the protocols is required, and the same service can be developed within 5 hours. The example demonstrates the applicability of the PhD framework prototype for developing value-added services. It shows that service developers can create their services in an easy and fast way and that the services fulfil the performance requirements for value-added telecommunication services.

9 Conclusion and Future Work

This chapter concludes the thesis. The achievements of the research work (refer to section 9.1) are summarised, the advantages of the solution are outlined (refer to section 9.2), the claims of novelty are presented (refer to section 9.3), the limitations of the research are discussed (refer to section 9.4), and suggestions and ideas for further research are proposed (refer to section 9.5).

9.1 Achievements of the Research

Based on the identified deficits, the aim of the performed research work was to find a method for description, an automated creation, execution, and provisioning of value-added telecommunication services. With the help of the methods found, the developers should be able to describe value-added services, even if they are not experts on all the relevant communication protocols that are required for a full-fledged service.

For the description of the services, BPEL is used as service description language. The service logic is described with BPEL, and the functionality is offered by the CBBs that are represented in BPEL as partner links. The service is generated automatically from the BPEL service description. To realise this automated creation of the services, a service execution environment was defined which is based on JAIN SLEE. The result of this research is an automated solution for the creation and provisioning of value-added telecommunication services.

Various other solutions in the field of service creation, execution, and provisioning were reviewed. Criteria for the required solution were defined and existing technologies and other research works were analysed regarding these criteria.

In the first step, current solutions for service creation and research projects dealing with this issue were analysed (refer to chapter 3). The existing solutions were evaluated regarding the criteria (i) support of a graphical development tool, (ii) abstraction from underlying protocols, (iii) support of new protocols in the service description, and (iv) the ability to define a broad range of value-added services.

The research revealed that BPEL fulfils the defined criteria and offers the possibilities required for the description of value-added services. Therefore, BPEL was chosen as service description language.

In the next step, the fields of technologies for service execution and service provisioning were analysed (refer to chapter 4). Individual advantages and disadvantages of each technology were shown, and research projects related to the topic were discussed. The technologies were evaluated regarding the criteria, which are (i) supported protocols, (ii) expandability, (iii) performance, (iv) service possibilities, (v) composition capability/reusability, and (vi) programming language. In summary of this step, with regard to the defined criteria, the JAIN SLEE environment was selected as base framework for the service execution and provisioning.

With BPEL, a wide variety of value-added services can be described. In this approach, BPEL was chosen for service development and JAIN SLEE for service execution. Therefore, no BPEL engine is needed for the service execution. A BPEL development tool can be used to create the service description. Instead of developing web services which can be composed to more complex services, value-added services will be created automatically from the service description. The created services will be executed in JAIN SLEE. From the results in chapter 3 and chapter 4, the requirements for the proposed framework were derived in section 5.1.

The gap between the service description in BPEL and the service execution in JAIN SLEE was researched in the next step. New ideas for a combination of the advantages from both selected technologies were discussed (refer to chapter 5). It was defined how a value-added service has to be described in BPEL. The service logic is described with the BPEL activities of a BPEL process. BPEL offers all logic elements that are required for a service. A value-added service also needs the possibility of communicating with other services, reading from databases, sending and receiving data, invoking other resources, and listening for other resources. These functionalities can be described with the BPEL partner links.

From the BPEL service description, a value-added service has to be generated. Two possible approaches were analysed, the Code Generator and the Runtime Service Composition. The advantages and disadvantages of both approaches were discussed, and the Runtime Service Composition approach was recommended for the framework.

In the next step, the general service structure was analysed. Several different service structure concepts were discussed. The advantages and disadvantages of each technology were analysed, and two technologies, the “Orchestration concept” and the “Choreography concept”, were chosen for the framework.

As result of the previous research steps, a framework for the creation and execution of the value-added services was defined (refer to chapter 6). This framework consists of the Service Creation Environment (SCE) that supports the description and management of the services, and of a Service Execution Environment (SEE) for the provisioning and execution of the value-added services.

The SCE consists of the Communication Building Blocks (CBBs), the service management tool, the graphical service description tool, the marketplace, and the service repository. With the service management tool, the service life cycle can be controlled and monitored. A BPEL development tool can be used for the graphical service description. The repository is a place where existing service descriptions can be stored. With the marketplace, new service descriptions and new resources can be acquired through the Internet.

CBBs define the available functionalities, resources, and supported protocols. With this concept, the new protocols, resources, and functionalities can be added to the framework. The CBBs provide a simple and comfortable possibility for the service designer to combine the service logic with the required functionalities. The service logic is described with the BPEL activities. The resources and the functionalities are described as BPEL partner links.

A CBB consists of multiple components. Some components are available within the SCE and some within the SEE. The SCE part of the CBBs consists of the partner links and the variable types. The partner links define methods, which correspond to the methods implemented in RCSBBs of the SEE. With these partner links, the functionalities can be described in the service description, which can be called from the service implementation in the SEE. The service designer can choose and configure the desired method that offers the required functionality from a BPEL partner link. When a CBB requires a variable type that is not available in the SEE, then this variable type is also part of the CBB.

The other parts of a CBB are the implementation of the functionalities, the adaptors to the resources, and the methods that call the new functionalities. The description of the functionalities and resources in BPEL is mapped to their implementation in the RCSBBs of the SEE.

The granularity of the functionality depends on the particular CBB. Fine-grained functionality and coarse-grained functionality can be provided to the service developer for describing the services. A CBB can hide the complexity of the underlying protocols from the developer, and the developer can concentrate on the logic of the service. In this case, detailed knowledge of the communication protocols is not required. On the other hand, it is also possible that a CBB offers fine-grained methods for a more detailed control of the protocol communication.

To support a new protocol, resource, or functionality, an appropriate CBB is required. CBBs can be provided by the framework developer, by third-party developers, or by the developers of a resource, functionality, or protocol. For the prototypical implementation, two CBBs were developed, the HTTP CBB and the SIP CBB.

The developed BPEL service description is analysed by the framework and the value-added service is generated from this description automatically.

A layered structure was defined for the SEE. This structure consists of three layers, a management layer, a service logic layer, and a resource connection layer. The management layer controls the framework and offers the possibilities to control the framework and the services via web interface. The service management also controls the creation and the execution, as well as the removal and reconfiguration of service instances. The service logic layer offers the components that are required to execute the service logic. The components of this layer are mapped from the activities in BPEL. The resource connection layer consists of the implementation of the functionality that was defined in the CBBs. The methods of the partner links are implemented in RCSBBs. RCSBBs in combination with the RAs offer the resources for the services. These RAs are also part of this layer and offer the protocol-specific communication.

The value-added services were examined in chapter 7. There, the service structure and the service life cycle were defined. A service consists of several components. Each component belongs to one of the defined layers of the SEE. The communication between the service components of a service instance on the one hand, and between components of the framework management on the other, is done via events.

In the last step (refer to chapter 8), the framework was evaluated regarding the defined requirements, and the proof of concept of the proposed framework was presented. For each requirement, it was analysed if it is fulfilled. The research prototype of the framework was introduced and the overall prototype functionality outlined. The research prototype was successfully adopted for a proof of concept of the proposed framework. Important service components were evaluated systematically. For each component, a minimal service description was developed, and the correspondent service was automatically generated and analysed.

Furthermore, a typical value-added service was developed by the help of the framework prototype. The service was described with a BPEL development tool and loaded into the service repository. With the service management tool, the service was triggered for creation and execution. The SEE creates and executes the service successfully. As proof of concept, the research prototype has demonstrated its applicability for developing and executing value-added telecommunication services.

9.2 Advantages of the Solution

This section analyses the most important advantages of the proposed solution.

Fast service development

Compared to the conventional development of value-added services, the development of services with the proposed framework is faster (refer to section 8.5.2). In the proposed framework, the value-added services are described with a BPEL (refer to section 3.6) development tool. In section 8.5.2, the service development with the proposed framework is compared with the conventional service development of the JSLEE framework (refer to section 4.4) in Java. With the proposed framework, the lines of code that are required for a service are significantly reduced. The BPEL process description is shorter than the Java code. In the example service in section 8.5.1, the length of the Java code is 252 lines. Compared to the BPEL code with a length of 196 lines, the Java code is quite longer. Furthermore, the conventionally developed JSLEE service requires additional support files for the deployment. The total lines of code of the conventional JSLEE service are 447, whereas the BPEL code consists of 196 lines.

The length of code can be influenced by the granularity of the CBB methods (refer to section 6.2.1) that are used in the service, but the total lines of code of a BPEL service description can vary. If more fine-grained services are developed, the service description is longer than a coarse-grained service but shorter than the Java code. In BPEL, only the functionality offered by the CBBs is described. The implementation of the CBB functionalities is done in Java.

The BPEL process descriptions can also be developed with a graphical service description tool. Compared to the conventional service description with Java, the graphical service description is faster than writing the Java code by hand.

The example service described in chapter 8.5.1 can be developed with Java within 3 days (24 hours). The service was developed conventionally with Eclipse and the JSLEE plugin for Eclipse in Java. The service development of the same value-added service with the PhD framework and the Eclipse BPEL designer takes only 5 hours.

In general, the description of services with BPEL is faster than developing the service with Java. The BPEL service description is more abstract than the Java code. The graphical service description with the BPEL activities and the partner links facilitates a more abstract service development. It is not required to describe all of the details, as would be necessary when developing a service in Java.

Easy service development

The development of conventional JSLEE services requires a deep knowledge of Java, of the JSLEE specification, and of the underlying protocols (refer to section 4.4). Therefore, only experts are able to develop value-added JSLEE services. JSLEE supports various protocols, and new protocols can be supported by adding a JSLEE resource adaptor for the new protocol. The developer has to understand how to use the protocol with the JSLEE resource adaptor.

The PhD framework facilitates the description of the services in BPEL. Many BPEL development tools offer a graphical user interface, which allows a graphical service development. The developer can use CBBs that encapsulate the functionality to communicate with other resources, e.g. web services or media servers. The CBBs hides the complexity of the underlying protocols and the communication with the resource adaptors from the developer. It is a middle layer which maps the description of the functionality to its implementation. The abstraction level of the CBB methods can vary in granularity. This offers a user-specific set of CBB methods, which can be used by the developer in BPEL (see next paragraph). The graphical service development together with the CBBs facilitates an easy development of value-added services (refer to section 8.4.2).

Fine-grained/coarse-grained

The CBB methods offer different levels of granularity for describing the functionalities (refer to section 6.2.1). The CBBs can provide coarse-grained methods, which facilitate a simple, more abstract usage of external resources, services, and other functionalities. A CBB method can encapsulate a complex functionality with complex protocol communication, which is hidden from the developer. Furthermore, CBBs can provide fine-grained methods for describing value-added services. This offers the possibility for a more detailed influence and more individuality in the service behaviour but the development of the service would require more knowledge and take more time. A CBB can provide multiple levels of granularity and the possibility to mix methods from different levels of granularity.

Support of the requirements of telecommunication services

The developed PhD framework supports the requirements of telecommunication services (refer to section 8.5.3) and the advantages of the JSLEE framework (refer to section 4.4). The advantages of JSLEE are flexibility, platform independence, low latency, and high throughput. The PhD framework is based on JSLEE, and JSLEE is specially developed for telecommunications. All elements of the developed framework consist of elements from the JSLEE framework. Therefore, the PhD framework, too, supports the requirements of telecommunication services.

Runtime service composition

The value-added services developed with the PhD framework consist of predefined elements. These elements are parts of the framework and are deployed together with the framework on the JSLEE AS server. The new services will not be deployed on the framework; they are orchestrated and instantiated at start time of the service from SBBs that are already deployed. This allows the monitoring and reconfiguration of the services at runtime (refer to section 5.3.2).

Conventional JSLEE services need to be deployed on the application server. Additional files for the service deployment are required. For new versions of the service, a new deployable unit with the service code and the deployment descriptors is required.

Expandability

The PhD framework is based on the extensible JSLEE framework (refer to chapter 6). JSLEE is based on Java, which is an extensible programming language. New resource adaptors can be added to the framework for supporting new protocols. Additionally, other than JSLEE, the PhD framework supports CBBs. These CBBs offer, in combination with the resource adaptors, a support of new protocols. Furthermore, they offer methods, resources, functionalities, and other services. New CBBs can be added to the framework by deploying them to the application server and using the CBB partner links for the service description in BPEL. The service developer can describe services which support these new functionalities by choosing the appropriate method from the partner link.

Service possibilities

Based on the functionality of this framework, a great number of services can be defined (refer to chapter 4). As described above, new resource adaptors and CBBs can be added to the framework. This offers a high number of possible services. In comparison to the other technologies described in chapter 4 the PhD framework offers nearly unlimited service possibilities, similar to JSLEE. The CBBs (refer to section 6.2.1) offer the possibility to use the new functionality in the BPEL service description as partner link. Therefore, it is possible to describe many value-added services with the framework. Apart from that, the composition of reusable service components is possible and will be described in the next paragraph.

Composition capability/reusability

The PhD framework consists of reusable components by which value-added services can be composed (refer to section 6.3), the logic components (LSBBs), and the resource components (RCSBBs/CBBs). All these service components are predefined and can be used to orchestrate new services. These reusable components can be utilised in multiple services and only need to be developed once.

In addition, the BPEL service description can be reused, and new services can be developed, based on already defined service descriptions.

Furthermore, services that are already developed can be embedded in a CBB. In this case, the required BPEL partner link methods for this CBB have to be defined to make the service functionality available in BPEL. This CBB can be used in future services to reuse the already developed services.

9.3 Claims of Novelty

This research work offers novel features in the fields of service description, service creation, service execution and service provisioning. This section summarises the most important novelties.

9.3.1 Novelties in Service Description

Describing value-added services with BPEL

This research work proposes BPEL for describing value-added services. Business processes are normally described with BPEL. There, a BPEL engine executes these business processes as web services. Other research projects also use BPEL to orchestrate telecommunication web services (OPUCE, Orchestration in web services and real-time communications, StarSCE, Orchestrated Execution Environment for Hybrid Services, and Orchestrated Execution Environment Based on JBI). In these projects, the value-added service can be controlled via web service interfaces through the BPEL process description, which is executed on a BPEL engine.

The approach taken in this research work uses BPEL to describe the service directly, without a BPEL engine and without using web services. The value-added service is created directly from the BPEL service description (refer to section 5.2).

BPEL offers the elements of a standard programming language and allows a wide range of service possibilities. In a normal business process, the BPEL process is used for the logic of the business process workflow. BPEL supports all elements that are required for a programming language including elements for parallel execution and loops. This thesis demonstrates the possibility to use BPEL as description language for the services (refer to section 5.2).

Graphical service description with existing BPEL description tools

This PhD work offers a possibility to describe value-added services with BPEL (refer to section 5.2.1). For BPEL, many graphical development tools are available which allow a graphical design of business process descriptions. With these tools, the service logic of a value-added service can be described graphically. The service developer can, therefore, choose which type of service creation environment is to be preferred – a graphical one or a textual one – or he can switch between both.

The BPEL process description is the input for the service creation

The output of the BPEL developer tool is the BPEL process service description (refer to section 5.3.2). A normal BPEL process is executed on a BPEL engine. In the research projects mentioned above (OPUCE, Orchestration in web services and real-time communications, StarSCE, Orchestrated Execution Environment Based on JBI), a BPEL engine is used to execute the telecommunication web services. In the research project "Orchestrated Execution Environment for Hybrid Services", a BPEL engine is deployed in a JSLEE resource adaptor and the BPEL process is executed on this engine. In this PhD work, the BPEL process is not executed on a BPEL engine. The BPEL service description is generated with a BPEL development tool and is uploaded into the developed PhD framework. The BPEL process description is parsed, and the value-added service is orchestrated and configured with the help of the service description.

Concept of the Communication Building Blocks (CBBs)

This PhD work introduces the concept of CBBs (refer to section 6.2.1). This concept defines a middle layer and describes the mapping between the implementation of the functionality and the description of the functionality. The functionality is implemented in JSLEE SBBs called RCSBBs. With these RCSBBs, the communication with the underlying JSLEE framework or the protocol-specific communication in combination with resource adaptors can be realised. The CBBs are mapped to the partner links in the BPEL service description. The service developer uses the partner link methods to describe the functionality of the service in BPEL. The CBB partner link methods offer an adaptive level of abstraction. The CBB developer can customize the granularity of the CBBs to meet the user's needs: he can use fine-grained CBB methods for a detailed service developing and course-grained CBB methods for a more abstract service description. When the CBB methods offer multiple levels of abstraction, the developer can choose the preferred level of abstraction.

Mapping of service logic (activities) to SEE components

This research work proposes the possibility to describe the service logic of the value-added services within BPEL (refer to section 5.2.2). The BPEL activities offer the required elements to describe the logic of the service. These activities support, for instance, loops, parallel execution, partner link calls, copy operations, sequences, and if-clauses. Furthermore, XPath is supported regarding complex transformations and expressions. Conventional value-added JSLEE-based services are developed with the Java programming language. Other research projects use BPEL to orchestrate their telecommunication web services and do not describe the value-added service directly.

Level of abstraction and abstraction from protocols

The CBBs define functionalities on top of the protocol-specific communication (refer to section 6.2.1). They define a middle layer between the implemented functionality and the service description. The CBBs are mapped to the partner links in BPEL. The granularity of the functionality depends on the particular CBB. Fine-grained functionality and coarse-grained functionality can be provided to the service developer for describing the services. This offers the possibility for a user-specific customisable level of abstraction. Depending on the knowledge and the requirements of the developer, a user-specific level of abstraction can be selected.

A CBB can hide the complexity of the underlying protocols from the developer, and the developer can concentrate on the logic of the service. In this case, detailed knowledge of the communication protocols is not required. On the other hand, it is also possible that a CBB offers fine-grained methods for a more detailed control of the protocol communication. In dependence of the functionality, which is offered by the CBB, the service developer can choose the preferred level of abstraction and can combine different levels of abstraction within the same service. A coarse-grained service description normally offers a more abstract way for describing a service and requires less understanding of the underlying protocols but reduces the possibility of establishing individual characteristics of the service.

9.3.2 Novelties in Service Creation, Service Execution and Service Provisioning

Services are automatically created from BPEL process descriptions

The developed BPEL process description is uploaded to the PhD framework. The value-added telecommunication service is generated automatically from the uploaded BPEL service description (refer to section 5.3.2). Therefore, the SCMSBB of the service parses the service description. It analyses the service description and orchestrates all LSBBs and RCSBBs which are required for the service. Each LSBB receives its individual part of the service description and can configure itself. For the service creation and configuration, the orchestration concept (refer to section 5.4.3) is used. The SCMSBB send events to all LSBBs and RCSBBs for their instantiation and configuration. When the SBBs are configured, they send an event to the

SCMSBB. With this event, they signal that they are ready for execution. From now on, all the elements of the service use the choreography concept (refer to section 5.4.4) for service execution. Each element knows how to execute its own part of the service workflow, and each element knows its communication partners.

In contrast to the conventional service creation, the services do not need a deployable unit and do not need to be compiled before. The service is orchestrated from already deployed components of the PhD framework. This architecture offers the possibility to monitor and reconfigure the service at runtime, for instance from a BPEL developer web interface.

The SEE of the framework is based on JSLEE

The framework is based on JSLEE (refer to chapter 6). All the defined components, the framework management components, the SCMSBBs, the LSBBs, and the RCSBBs of the framework, consist of JSLEE SBBs. A layered structure is defined on top of the JSLEE component container. The three layers are a management layer, a service logic layer, and a resource connection layer.

The framework management SBBs and the SCMSBBs are part of the management layer, whereas the LSBBs belonging to the service logic layer and the RCSBBs are part of the resource connection layer. Therefore, the management, the execution and the communication with resource adaptors and other resources is done by SBBs.

The result of the service creation is a value-added telecommunication service, which also consists of components of the JSLEE framework and therefore fulfils the requirements of telecommunication services.

Other research projects which use BPEL execute these services on a BPEL engine. They use BPEL to orchestrate their telecommunication web services. The services that are orchestrated with the BPEL process have to be developed conventionally. The development of value-added services with BPEL is not within the scope of these projects, because a BPEL engine does not fulfil the requirements of telecommunication web services.

9.4 Limitations of the Research

Although the overall objectives of the research project had been met, some decisions had to be taken which resulted in limitations imposed on the work. Those decisions were caused by practical reasons, or were made to delimit the considered research project from related fields of study which could not be fully covered by this research due to generally given time scope limitations for the accomplishment of research degree studies. The key limitations are summarised below.

1. In this research, BPEL was selected as service description language, which consists of numerous elements. For the prototype, not all of the possible BPEL language elements and concepts are implemented. This also affects the BPEL activities. For the prototypical framework implementation, only those BPEL activities are implemented that are required for the proof of the concept. The rest of the activities can be considered for a real-world implementation. Other BPEL concepts, e.g., the BPEL fault-handling concept and the correlations concept have not been considered.

2. Furthermore, the prototype does not analyse all parts of the BPEL process. This affects, e.g., the import part and the namespace declarations, parts which are not relevant for the prototype.
3. BPEL supports the x-path language; this language was not implemented within the prototype. However, to support conditions for the while- and if-activities and to define expressions within the Assign activity, a limited set of conditions and expressions are supported.
4. For the evaluation of this work, two CBBs were implemented, a CBB for HTTP functionality and a CBB for SIP functionality. Both of the CBBs only support a limited range of the SIP and HTTP functions. Only the functions that are required for the evaluation have been implemented.
5. The marketplace interface, which is described in the architecture overview, was not implemented. For the prototypical implementation, it was not necessary to demonstrate how to acquire CBBs or service descriptions from the Internet. The service descriptions can be transferred from a computer's file system into the repository by using a web interface. The marketplace is required for a real-world product but is not relevant for the prototype.
6. The architecture offers the possibility to support different service description parsers. However, this is a possibility and not a requirement. Therefore, the prototype only supports BPEL as service description language. How another service description can be supported by the framework, may be part of further research.

Despite these limitations, the research project made valid contributions to knowledge and provided sufficient proof of concept for the proposed approaches.

9.5 Suggestions and Scope for Future Work

This research project extended the field of automated service creation. However, a number of areas for future work can be identified upon the results of this project. Some of these areas have already been mentioned in previous chapters. Possible areas for future work, enhancements, and improvements are:

1. Further research may address the issue how multiple service description languages can be supported. Regarding the service description, this research work focused on BPEL. Other service description languages can be supported by adding service description parsers for these new service description languages.
2. Another extension could be an interactive web interface for graphical service development, monitoring, managing, and runtime manipulation of a service. This work requires a service description from a BPEL developer tool. The service description is uploaded to the service repository with a web interface. The web interface is also used for the management of the service. In the next step, the services, too, can be developed with a web interface. The idea is to develop the service within a graphical web interface or to load an existing service from the repository into the graphical web interface. The graphical representation of the service in the web interface could be the graphical BPEL process. This graphical representation may display the status and the configuration of each service component of a service instance at runtime. With this web interface, it could be possible to develop the services like in a

BPEL development tool. The service could be started, monitored, and manipulated directly from this web interface.

3. In this research work, the external resources and functionalities are available through CBBs. Partner links are used to call resources and functionalities in the service description. In the same way, an already developed service could be used in a new service description. To make the developed services available for the usage within the service description of another service, the partner link description and, e.g., the RCSBB would have to be developed manually. An automated creation of the RCSBBs and the BPEL partner link description could be a comfortable way to develop services which use the functionality of already developed services. This step could result in a new CBB that can be the base for new service.
4. In the proposed concept, each activity is mapped to one LSBB in the service execution layer. In a further step it could be researched which of the activities could be integrated to other LSBBs. Maybe the “if” condition, the assign or the loops could be integrated to other LSBBs. This optimisation might reduce the number of SBBs per service instance and the amount of event communication between the service components during the event execution.

References

1. 3GPP TS 23.228 TS 23.228 (2006), Technical Specification, "IP Multimedia Subsystem (IMS); Stage 2 (Release 5)", 3GPP

2. 3GPP TS 23.228 (2013), Technical Specification, "Digital cellular telecommunications system (Phase 2+); Universal Mobile Telecommunications System (UMTS); LTE; IP Multimedia Subsystem (IMS); Stage 2 (Release 11)", 3GPP

3. 3GPP (2014), "THE Mobile Broadband Standard", http://www.3gpp.org (last visited: 2014-03-24)

4. Abarca, C.; Bennett, A.; Moerdijk A. J.; Unmehopa, M. (2002), "Parlay/OSA: an open API for service development", 3GPP, http://www.3gpp.org/ftp/tsg_sa/WG3_Security/TSGS3_22_Bristol/Docs/PDF/S3-020126.pdf (last visited: 2014-06-14)

5. ANT (2010), "The Apache ANT Project", http://ant.apache.org/ (last visited: 2010-12-28)

6. Autili, M.; Berardinelli, L.; Cortellessa, V.; Di Marco A.; Di Ruscio D.; Inverardi, P.; Tivoli, M. (2007), "A Development Process for Self-adapting Service Oriented Applications", in proceeding of ICSOC 2007 (442-448), Vienna, Austria

7. Bakker, J. L.; Jain, R. (2002), "Next Generation Service Creation Using XML Scripting Languages", Proc. IEEE Intl. Conf. on Comm. (ICC), IEEE

8. Baravaglio, A.; Licciardi, C. A.; Venezia, C. (2005), "Web service applicability in telecommunication service platforms", IEEE International Conference on Next Generation Web Service Practices 2005 (NWeSP 2005), ISBN: 0-7695-2452-4, 22-26, IEEE

9. Baresi, L.; Di Nitto, E.; Ghezzi, C. (2006), "Toward Open-World Software: issues and challenges", IEEE Computer, Volume 39, No. 10: 36-43, IEEE

10. Becker (2015), "CPL-Editor", X-ING project, Lehrstuhl Systemarchitektur, Institut für Informatik, Humbold-Universität Berlin, http://www2.informatik.hu-berlin.de/~xing/CPLEditor/ (last visited: 2015-05-13)

11. Bessler, S.; Zeiss, J.; Gabner, R.; Gross, J. (2007), "An Orchestrated Execution Environment for Hybrid Services", In Proc. Kommunikation in verteilten Systemen (KIVS), Bern, Springer

12. Bo, Ye; Da, Zhu; Yang, Zhang; Junliang, Chen (2009), "The Design of an Orchestrated Execution Environment Based on JBI", Computer Science-Technology and Applications, IFCSTA '09, International Forum on Computing & Processing (Hardware/Software), 2009, pp. 367-371, IEEE

13. Burgy L.; Consel C.; Latry F.; Lawall J.; Palix N.; Réveillère L. (2006), “Language Technology for Internet-Telephony Service Creation”, IEEE International Conference on Communications 2006, Instanbul, ISBN: 1-4244-0355-3

14. Chen L.; Wassermann B.; Emmeric W.; Foster H. (2006), “Web Service Orchestration with BPEL”, Proceedings of the 28th international conference on Software engineering, Shanghai 2006, pp. 1071-1072, 2006, ISBN:1-59593-375-1

15. Cipolla, D.; Cosso, F.; Demartint, M.; Drewniok, M.; Moggia, F.; Rendetore, P.; Sienel, J. (2007), “Web Services Based Asynchronous Service Execution Environment” Service-Oriented Computing - ICSOC 2007 Workshops, pp. 304-316, Vienna, Austria, 2007, Springer Berlin Heidelberg, 2009, ISBN: 978-3-540-93851-4

16. Curbera, F.; Goland, Y.; Klein, J.; Leymann, F.; Roller, D.; Thatte, S.; Weerawarana, S. (2002), “Business Process Execution Language for Web Service (BPEL4WS) 1.0”, BEA Corp., IBM Corp. and Microsoft Corp.

17. Detecon (2007), “Abschlussbericht Detecon – Services in NGN” (translated title: “Final Report Detecon – Services in NGN”), University of Applied Sciences Frankfurt am Main and Detecon International GmbH

18. Drögenhorn, O.; König, I.; Belaunde, M.; Le-Jeune, G.; Cupillard, J.; Kovacs, E. (2008) “Professional and End-User-Driven Service Creation in the SPICE platform”, World of Wireless, Mobile and Multimedia Networks (WoWMoM 2008), Newport Beach, 2008, 978-1-4244-2099-5, IEEE

19. Eclipse (2013), “BPEL Designer Project”, The Eclipse Foundation, http://www.eclipse.org/bpel/ (last visited: 2013-10-17)

20. Eichelmann, T.; Fuhrmann, W.; Trick, U.; Ghita, B. (2008), “Creation of value-added services in NGN with BPEL”, Internal Publication, In (Bleimann, U.; Dowland, P. S.; Furnell, S. M.; Grout, V. M.) Proceedings of the Fourth Collaborative Research Symposium on Security, Elearning, Internet and Networking (SEIN 2008), Wrexham, UK 2008. Centre for Security, Communications and Network Research, University of Plymouth, Plymouth, UK, 2008, ISBN: 978-1-84102-196-6, pp186–193

21. Eichelmann, T.; Fuhrmann, W.; Trick, U.; Ghita, B. (2009), “Support of parallel BPEL activities for the TeamCom Service Creation Platform for Next Generation Networks”, Internal Publication, In (Bleimann, U., Dowland, P.S., Furnell, S.M., Grout, V.M.) Proceedings of the Fifth Collaborative Research Symposium on Security, E-learning, Internet and Networking (SEIN 2009), Darmstadt 2009. Centre for Security, Communications and Network Research, University of Plymouth, Plymouth, UK, ISBN: 978-1-84102-236-9, pp69–80

22. Eichelmann, T.; Fuhrmann, W.; Trick, U.; Ghita, B.V. (2010) „Enhanced Concept of the TeamCom SCE for Automated Generated Services Based on

JSLEE“, Proceedings of the Eighth International Network Conference (INC 2010), Heidelberg, Germany, 8-10 July, ISBN: 978-1-84102-259-8, pp75-84

23. Eichelmann, T.; Fuhrmann, W.; Trick, U.; Ghita, B.V. (2011) “Discussion on a Framework and its Service Structures for generating JSLEE based Value-Added Services“, Proceedings of the Fourth International Conference on Internet Technologies & Applications 2011 (ITA 2011), Wrexham, UK, 2011, ISBN: 978-0-946881-68-0, p 169-176

24. Eichelmann, T.; Fuhrmann, W.; Trick, U.; Ghita, B.V. (2012), “A JSLEE based Service Creation and Service Delivery Framework for value-added services in Next Generation Networks”, Proceedings of the 3rd International Conference on Internet and Applications 2012 (ITAP 2012), Wuhan, China, 2012, ISBN: 978-1-4577-1575-4

25. ETSI TR 180 001 V1.1.1 (2006), Technical Report, “Telecommunications and Internet converged Services and Protocols for Advanced Networking (TISPAN); NGN Release 1; Release definition”, ETSI

26. ETSI TS 122.001 V8.0.0 (2009), Technical Specification, “Principles of circuit telecommunication services supported by a Public Land Mobile Network (PLMN)”, ETSI

27. ETSI TS 122.101 V9.3.0 (2009), Technical Specification, “Service aspects; Service principles”, ETSI

28. ETSI TS 122.105 V8.4.0 (2008), Technical Specification, “Services and service capabilities”, ETSI

29. ETSI TS 122.228 V8.6.0 (2009), Technical Specification, “Service requirements for the Internet Protocol (IP) multimedia core network subsystem (IMS)”, ETSI

30. EURESCOM 0241-1109 (2001), “Next Generation Networks: the service offering standpoint”, Eurescom

31. Falcarin, P.; Licciardi, C. A. (2003a), “Analysis of NGN services creation technologies” IEC Annual Review of communications, volume 56

32. Falcarin, P.; Licciardi, C. A. (2003b), “Technologies and Guidelines for Service Creation in NGN” 8th ITU-IEEE ICIN, Bordeaux

33. Fan, Q.; Glitho, R.; Khoumsi, A. (2006), “Creation of internet-telephony services using Siplet technology”, Proceedings of the International Conference on Telecommunications and International Conference on Internet and Web Applications and Services (AICT-ICIW 2006), IEEE, 0-7695-2522-9/06, 2006, Heidelberg, Germany, 8-10 July, ISBN: 978-1-84102-259-8, pp75-84

34. Fraunhofer SIT (2014), “MAMS Multi-Access, Modular-Services”, https://www.sit.fraunhofer.de/en/offers/projekte/mams/ (last visited: 2014-03-17)

35. Freese, B.; Stein, H.; Magedanz, T.; Dutkowski, S. (2007), “Multi-access modular-services framework - supporting SMEs with an innovative service

creation toolkit based on integrated SDP/IMS infrastructure", 11th International Conference on Intelligence in Service Delivery Networks, ICIN 2007. Proceedings. CD-ROM : Bordeaux, 8-11 October 2007 Bordeaux

36. Glitho, R. H.; Poulin, A.; Khendek, F. (2002), "A High Level Service Creation Environment for Parlay in a SIP Environment", ICC 2002 IEEE International Conference, pp 2008-2013

37. Glitho, R. H.; Khendek, F.; De Marco, A. (2003), "Creating Value Added Services in Internet Telephony: An Overview and a Case Study on a High-Level Service Creation Environment", Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions, pp 446 - 457

38. Görtz, M. (2005), "Effiziente Echtzeit-Kommunikationsdienste durch Einbeziehung von Kontexten" (translated title: "Efficient Real-time Communication Services through the using of Contexts"), Dissertationsschrift TU Darmstadt

39. Haiges, S. (2005), "Einführung in die JAIN/SLEE-Technologie" (translated title: "Introduction to the JAIN/SLEE Technology"), it-republik

40. Hammerschall, U. (2005), "Verteilte Systeme und Anwendungen" (translated title: "Distributed Systems and Applications"), Pearson

41. IBM Corporation (2012), "SOA + Telecommunications = TelcoML!: An introduction to the TelcoML design standard, an UML Profile for Integrated Telecom", IBM

42. IBM WebSphere (2015), "WebSphere Voice Toolkit V6.2", First published by IBM developerWorks at http://www.ibm.com/developerWorks/websphere/downloads/voicetoolkit.html All rights retained by IBM and the author(s) (last visited: 2015-05-13)

43. IETF (2001a), Internet Draft "An Application Server Component Architecture for SIP", IETF

44. IETF (2001b), Internet-Draft, "SIP Servlet API Extensions", IETF

45. IETF (2001c), Internet Draft, "A Service Creation Markup Language for Scripting Next Generation Network Services (SCML)", IETF

46. IETF (2005), Internet-Draft "LESS: Language for End System Services in Internet Telephony", IETF

47. IETF RFC 3050 (2001), Request for Comments, "Common Gateway Interface for SIP", IETF

48. IETF RFC 3362 (2002), Request for Comments, "Real-time Facsimile (T.38) - image/t38 MIME Sub-type Registration", IETF

49. IETF RFC 3880 (2004), Request For Comments, "Call Processing Language (CPL): A Language for User Control of Internet Telephony Services", IETF

50. IETF RFC 4267 (2005), Request For Comments, "The W3C Speech Interface Framework Media Types: application/voicexml+xml, application/ssml+xml,

application/srgs, application/srgs+xml, application/ccxml+xml, and application/pls+xml”, IETF

51. IETF RFC 5022 (2007), Request for Comments, “Media Server Control Markup Language (MSCML) and Protocol”, IETF

52. IETF RFC 5707 (2010), Request for Comments, “Media Server Markup Language (MSML)”, IETF

53. IETF RFC 6120 (2011), Request For Comments “Extensible Messaging and Presence Protocol (XMPP): Core”, IETF

54. IPCC (2002), “Reference Architecture”, International Packet Communications Consortium

55. ITIL (2014), “Glossar”, http://www.itil.org/en/glossar/glossarkomplett.php (last visited: 2014-03-03)

56. IST NGN Initiative (2002), “NGNI Roadmap 2002”, NGNI http://www.ngni.org

57. istworld (2014), “Open platform for user-centric service creation and execution”, http://www.ist-world.com/ProjectDetails.aspx?A=1&DataSelectionDataType=Project&DataSelectionSelectionType=classification&DataSelectionSourceDatabaseId=7cff9226e582440894200b751bab883f&FindClassificationEntityName=Project&FindClassificationCategoryName=Top%2FScience%2FChemistry%2FNuclear_Magnetic_Resonance%2FCommercial_Companies%2FMagnet_Service&FindClassificationDisplaySubs=true (last visited: 2014-06-16)

58. ITU-T I.112 (1993), Recommendation, “Integrated Services Digital Networks (ISDN) General Structure”, ITU-T

59. ITU-T I.210 (1993), Recommendation, “Principles of Telecommunication Services supported by an ISDN and the means to describe them”, ITU-T

60. ITU-T I.211 (1993), Recommendation, “B-ISDN Service aspects”, ITU-T

61. ITU-T Q.1200 (1997), Recommendation, “General series Intelligent Network Recommendation structure”, ITU-T

62. ITU-T Y.2001 (2004), Recommendation, “General overview of NGN”, ITU-T

63. JDeveloper (2014), “Oracle JDeveloper”, http://www.oracle.com/technetwork/developer-tools/jdev/overview/index.html (last visited: 2014-03-18)

64. Jouve, W.; Palix, N.; Consel, C.; Kadionik, P. (2008), “A SIP-based Programming Framework for Advanced Telephony Applications”, In Principles, Systems and Applications of IP Telecommunications Services and Security for Next Generation Networks, IPTCOM 2008, Heidelberg, Germany

65. JSR 3 (2000), Java Specification Requests, “JSR-000003 Java^{TM} Management Extensions (JMX) v1.0 Specification (Final Release)**”,** Sun Microsystems, Inc.

66. JSR 16 (2000), Java Specification Requests, “JSR-000016 J2EE^{TM} Connection Architecture (JCA) v1.0 Specification (Final Release)**”,** Sun Microsystems, Inc.

67. JSR 19 (2001), Java Specification Requests, “JSR-000019 Enterprise JavaBeansTM v2.0 Specification (Final Release)**”,** Sun Microsystems, Inc.

68. JSR 21 (2002), Java Specification Requests, “JSR-000021 Java^{TM} Call Control API Specification (JCC^{TM}) v1.1 Specification (Final Release)**”, S**un Microsystems, Inc.

69. JSR 22 (2004), Java Specification Requests, “JSR-000022 Specification, JAIN SLEE 1.0 Specification, Final Release**”,** Sun Microsystems Inc., Open Cloud

70. JSR 116 (2003), Java Specification Requests, “JSR-000116 Specification, “SIP Servlet API, Final Release”, Sun Microsystems, Inc., dynamicsoft inc.

71. JSR 160 (2003), Java Specification Requests, “JSR-000160 Java^{TM} Management Extensions (JMX) Remote API 1.0 (Final Release)**”,** Sun Microsystems, Inc.

72. JSR 208 (2005), Java Specification Requests, “Java™ Business Integration 1.0 (JBI 1.0)”, Sun Microsystems, Inc.

73. JSR 240 (2008), Java Specification Requests, Final Release, “JAIN SLEE (JSLEE) 1.1”, Sun Microsystems, Inc

74. JSR 255 (2008), Java Specification Requests, “JSR-000255 Java^{TM} Management Extensions (JMX^{TM}) v2.0 Specification (Early Draft Review)**”, S**un Microsystems, Inc.

75. JSR 289 (2008), Java Specification Requests, JSR-000289 Specification, Final Release, **“**SIP Servlet Specification, version 1.1**,** Sun Microsystems, Inc., Oracle, BEA

76. Juric, M. B. (2014), “A Hands-on Introduction to BPEL”, In: Developer: J2EE Web Services, http://www.oracle.com/technetwork/articles/matjaz-bpel1-090575.html (last visited: 2014-03-14), Oracle

77. Kanbach, A.; Körber, A. (1999), “ISDN – Die Technik” (translated title: “ISDN – The Technology”), Hüthig

78. Keiser, J.; Kriengchaiyapruk, T. (2008), “Bringing Creation of Context-Aware Mobile Services to the Masses”, In: IEEE SOA Industry Summit (SOAIS 2008), Hawaii, USA

79. Kuthan, Jiri (2000), “Moving Telephony from Intelligent Networks to Dumb Net-works”, SIP2000 Konferenz Paris

80. Lasch, R.; Ricks, B.; Tönjes, R. (2009a), “Service Creation Environment for Business-to-business Services”, 4th International IEEE Workshop on Service Oriented Architectures in Converging Networked Environments, Bradford, UK

81. Lasch, R.; Ricks, B.; Tönjes, R. (2009b), "Konzept eines BPEL zu JSLEE Compilers auf Basis wieder-verwendbarer Kommunikationsbausteine" (translated title: "Concept of a BPEL to JSLEE Compiler based on re-usable Communication Building Blocks"), In (Tönjes, R., Roer, P.) Mobilkommunikation – Technologien und Anwendungen – Vorträge der 14. ITGFachtagung from 13th to 14th May 2009 in Osnabrück, Osnabrück 2009, VDE, Berlin, 2009. ISBN 978-3-8007-3164-0k

82. Latry F.; Mercadal J.; Consel C. (2007), "Staging Telephony Service Creation: A Language Approach", In Principles, Systems and Applications of IP Telecommunications, IPTComm, New-York, NY, USA, ACM Press

83. Lehmann, A.; Trick, U.; Oehler, S. (2007), "NGN und Mehrwertdienste – Technische Lösungen" (translated title: "NGN and Value-added Services – Technical Solutions"), ntz Nachr.-tech. Z. 60 (2007) H. 7-8, S. 30 –33

84. Lehmann, A.; Trick, U.; Oehler, S. (2008a), "NGN und Mehrwertdienste – Geschäftsmodelle und Szenarien" (translated title: "NGN and Value-added Services – Business Models and Scenarios"), ntz Nachr.-tech. (2008) H. 1, S. 22 –25

85. Lehmann, A.; Eichelmann, T.; Trick, U.; Lasch, R.; Tönjes, R. (2009), "TeamCom: A Service Creation Platform for Next Generation Networks", In (Perry, M.; Sasaki, H.; Ehmann, M.; Bellot, G. O.; Dini, O.) The Fourth International Conference on Internet and Web Applications and Services (ICIW 2009), Venice 2009, IEEE Computer Society, Los Alamitos, CA, USA, 2009, ISBN 978-0-7695-3613-2

86. Lehmann, A. (2010), "Optimisation of SIP-based peer-to-peer communication under special consideration of provisioning and composition of distributed value-added services", University of Plymouth, United Kingdom

87. Licciardi, A. C.; Falcarin, P. (2003), "Technologies and Guidelines for Service Creation in NGN", exp Volume 3 -- n. 4

88. Lin, L.; Lin, P. (2007), "Orchestration in Web Services and Real-Time Communications", IEEE Communication Magazine

89. Lu, H.; Zheng, Y.; Sun, Y. (2008), "The Next Generation SDP Architecture: Based on SOA and Integrated with IMS", Second International Symposium on Intelligent Information Technology Application (IITA'08), Volume 3, pp. 141-145, IEEE

90. Magedanz, T.; Sher, M. (2006), "IT-based Open Delivery Platforms for Mobile Networks: From CAMEL to the IP Multimedia System", The handbook of mobile middleware, Boca Raton, CRC Press, 2006, ISBN 978-0-8493-3833-5, pp. 999-1034

91. Magedanz, T.; Blum, N.; Dutkowski, S. (2007), "Evolution of SOA concepts in Telecommunications", IEEE Computer Magazine

92. MAMS (2010), “Multi-Access, Modular-Services Framework (MAMS)”, Fraunhofer FOKUS, BMBF-Project, http://www.mams-platform.net (last visited: 2010-12-28)

93. Maretzke, M. (2005), “JAIN SLEE Technology Overview”, http://www.maretzke.de/pub/lectures/jslee_overview_2005/JSLEE_Overview_2005.pdf (last visited: 2014-06-24)

94. Maretzke, M.; Haiges, S.; Bröcker, C. (2005), “Ereignisorientierte Komponenten mit JAIN SLEE”(translated title: “Event-oriented Components with JAIN SLEE”), Java Spektrum 05/2005

95. Martins, E. (2011), “Mobicents JAIN SLEE SIP Wake Up Example User Guide, Revision 2.0”, http://docs.jboss.org/mobicents (last visited: 2014-06-18), Oracle

96. Mobicents (2014), “The Open Source JAIN SLEE”, http://www.mobicents.org/slee/intro.html (last visited: 2014-01-18)

97. Moriana Group (2004), “SDP Thought Leader Community: Service delivery Platforms and Telecom Web Services – An Industry Wide Perspective”, Report on Service delivery Platforms, The Moriana Group

98. Moriana Group (2013), “Moriana on SDP 2.0: Service Delivery Platforms – definition and evolution”, Technology Article, Available at: http://www.morianagroup.com/index.php?option=com_content&view=article&id=148&Itemid=233 (last visited: 2014-03-15)

99. Mulvenna, M.; Valetto, G.; Hayden, C.; McConnel, R.; Lawrynowicz, A.; Baumgarten, M.; (2008), “The Potential for Autonomic Service Delivery Platforms”, Proceedings of the 2008 International Conference on Communication in Computing (CIC 2008), CSREA Press, USA, pp. 188-194

100. OASIS Standard (2006), “Reference Model for Service Oriented Architecture 1.0”, OASIS

101. OASIS (2007), “Web Services Business Process Execution Language Version 2.0”, OASIS

102. ODE (2013), Apache ODE (Orchestration Director Engine), http://ode.apache.org/ (last visited: 2013-03-4)

103. OMA (2004), “OMA Service Environment Approved Version 1.0 – 07 Sep 2004”, OMA-Service_Environment-V1_0-20040907-A, OMA

104. OMA (2013), “Open Mobile Alliance”, http://www.openmobilealliance.org (last visited: 2012-12-28)

105. OMG (2008), “UML Profile and Metamodel for Voice-based Applications Specification (VoicP)”, Version 1.0

106. OMG (2010), “BPMN 2.0 by Example”, Version 1.0 (non-normative)

107. OMG (2011a), "Unified Modeling Language (OMG UML), Infrastructure", Version 2.4.1

108. OMG (2011b), "Business Process Model and Notation (BPMN) Specification", Version 2.0

109. OMG (2012a), "Common Object Request Broker Architecture (CORBA) Specification", Version 3.3

110. OMG (2012b), "Service oriented architecture Modeling Language (SoaML) Specification", Version 1.0.1

111. OMG (2013), "UML Profile for Advanced and Integrated Telecommunication Services (TelcoML) Specification", Version 1.0

112. OpenCloud (2013), "Rhino Visual Service Architect", www.opencloud.com/products/rhino-VSA/ (last visited: 2014-08-06)

113. OPUCE (2010), "Open Platform User-centric Service Creation and Execution“, http://www.redhat.com/solutions/telco/industry/opuce.html (last visited: 2010-12-28) or http://www.opuce.tid.es/ (not accessable on 2010-12-28)

114. Orthman, F. D. (2003), "Softswitch – Architecture for VOIP", McGraw-Hill

115. Ottinger, J. (2008), "What is an App Server?", http://www.theserverside.com/news/1363671/What-is-an-App-Server (last visited: 2015-03-09)

116. P1109 (2001), "Next Generation Networks: the service offering standpoint" Eurescom

117. Parlay (2010), "Parlay Informationen" (translated title: "Parlay Information"), http://www.parlay.org (last visited: 2010-12-28)

118. PhonerLite (2014), "PhonerLite", http://www.phonerlite.de/ (last visited: 2014-05-25)

119. Rosenberg, J.; Lennox, J.; Schulzrinne, H. (1999), "Programming Internet Telephony Services", IEEE Internet Computing Magazine

120. ServiceMix (2013), Apache ServiceMix, http://servicemix.apache.org/ (last visited: 2013-03-1)

121. Sienel, J.; Martin, A. L.; Zorita, C. B.; Goix, L. W.; Reol, A. M.; Martinez, B. C. (2009), "OPUCE: A telco-driven service mash-up approach", Bell Labs Technical Journal (Volume:14, Issue1), Alcatel-Lucent, 2009, ISS: 1089-7089, IEEE

122. SPICE (2013), "Service Platform for Innovative Communication Environment (SPICE)", European IST-FP6 project, http://www.ist-spice.org (last visited: 2013-12-28)

123. SPICE NEC (2013), "SPICE Service Platform for Innovative Communication Environment", NEC Europe Ltd. European IST-FP6 project

http://uk.nec.com/en_GB/emea/about/neclab_eu/projects/spice.html (last visited: 2013-10-13)

124. SPL (2013), “SPL (Session Processing Language – A DSL for IP telephony services)”, http://phoenix.inria.fr/software/past-projects/spl (last visited: 2013-07-08)

125. Stallings, W. (1999), “ISDN and Broadband ISDN with Frame Relay and ATM”, Pearson Education

126. Steffen, B.; Margaria, T.; Nagel, R.; Jörges, S.; Kubczak, C. (2006),“ Model-Driven Development with the jABC”, IBM Haifa Verification Conf. (HVC 2006), LNCS 4383, Springer-Verlag, pp.92-108

127. Steffen, B.; Narayan, P. (2007), “Full Life-Cycle Support for End-to-End Processes”, IEEE Computer Magazine

128. Sun Microsystems; Open Cloud (2003), “JAIN™ SLEE Tutorial Serving the Developer Community”, Sun and Open Cloud

129. TeamCom (2010), “TeamCom- IMS- or P2P-based Service Provisioning and Creation for Customer Tailored Communication Processes”, http://www.ecs.hs-osnabrueck.de/24009.html (last visited: 2010-12-28)

130. Trick, U., Weber, F. (2006), “Mobilität und Next Generation Networks (NGN)”, Band 1 “VDE Kongress 2006 Aachen – Innovations for Europe”, S.181-186, October 2006

131. Trick, U.; Weber, F. (2007), “SIP, TCP/IP und Telekommunikationsnetze” (translated title: “SIP, TCP/IP and Telecommunication Networks”), Oldenbourg, ISBN: 3-486-27529-1

132. Trick, U.; Weber, F. (2009), “SIP, TCP/IP und Telekommunikationsnetze (4th edition)” (translated title: “SIP, TCP/IP and Telecommunication Networks (4th edition)”), Oldenbourg, Munich, Germany, ISBN: 3-486-59000-5

133. Van Den Bossche B.; De Turck F.; Dhoedt B. and Demeeste P. (2006), “Enabling Java-based VoIP backend platforms through JVM performance tuning” Department of In-formation Technology (INTEC)

134. Venezia, C.; Licciardi, C.A. (2006), “Communication Web Services Composition and Integration”, IEEE International Conference on Web Services 2006 (ICWS ‘06), ISBN: 0-7695-2669-1, 18-22 Sept. 2006, IEEE

135. W3C (1999), “XML Path Language (XPath) Version 1.0”, W3C

136. W3C (2004a), “Web Services Architecture”, W3C

137. W3C (2004b), Recommendation, “Voice Extensible Markup Language (VoiceXML) 2.0”, W3C

138. W3C (2007a), Recommendation, “Voice Extensible Markup Language (VoiceXML) 2.1”, W3C

139. W3C (2007b), Recommendation, “Web Service Description Language (WSDL) Version 2.0 Part 1: Core Language”, W3C

140. W3C (2007c), Recommendation, “SOAP Version 1.2 Part 0: Primer (Second Edition)”, http://www.w3.org/TR/2007/REC-soap12-part0-20070427/ (last visited: 2014-06-24) , W3C

141. W3C (2008), Recommendation, “Extensible Markup Language (XML) 1.0 (Fith Edition)”, W3C

142. W3C (2011), Recommendation, “Voice Browser Call Control: CCXML”, Version 1.0, W3C

143. W3C (2013), Last Call Working Draft, “State Chart XML (SCXML): State Machine Notation for Control Abstraction”, W3C

144. Wu X.; Schulzrinne H. (2003), “Programmable end system services using SIP”, IEEE International Conference on Communications, May, 2003, 2nd New York Metro Area Networking Workshop, IEEE

145. Wu X.; Schulzrinne H. (2007), “Handling Feature Interactions in the Language for End System Services”, Computer Networks: The International Journal of Computer and Telecommunications Networking, Volume 51 Issue

Appendix A – Abbreviations

3GPP	Third Generation Partnership Project
3PCC	Third Party Call Control
A	
AAA	Authentication, Authorization, and Accounting
AC	Activity Context
API	Application Programming Interface
AS	Application Server
ASDL	Asymmetric Digital Subscriber Line
ATM	Asynchronous Transfer Mode
B	
B2BUA	Back-to-Back User Agent
BMBF	Federal Ministry of Education and Research
BPEL	Business Process Execution Language
BPMN	Business Process Model and Notation
BRAS	Broadband Remote Access Server
BS	Base Station
BSS	Business Support System
C	
CA	Client Application
CAMEL	Customised Applications for Mobile networks Enhanced Logic
CAP	CAMEL Application Part
CBB	Communication Building Block
CCXML	Call Control Extensible Mark-up Language
CGI	Common Gateway Interface
CN	Core Network
CORBA	Common Object Request Broker Architecture

CPL	Call Processing Language
CS	Call Server
CSCF	Call Session Control Function
CSE	Customised Application for Mobile Network Enhanced Logic Service Environment

D

DSL	Domain-Specific Language
DTMF	Dual-tone multi-frequency

E

EJB	Enterprise JavaBeans
ESB	Enterprise Service Bus
ETSI	European Telecommunications Standards Institute

G

GSM	Global System for Mobile communications
GUI	Graphical User Interface
GW	Gateway

H

HSS	Home Subscriber Server
HTML	Hypertext Mark-up Language
HTTP	Hypertext Transfer Protocol

I

IDE	Integrated Development Environment
IETF	Internet Engineering Task Force
IIOP	Inter Inter-Orb Protocol
IM	Instant Messaging
IMS	IP Multimedia Subsystem
IM-SSF	IP Multimedia-Service Switching Function

IN	Intelligent Network
IP	Internet Protocol
ISDN	Integrated Services Digital Network
ISONI	Intelligent Service Orientated Network Infrastructure
ITU-T	International Telecommunication Union - Telecommunication Standardization Sector

J

JAIN	Java APIs for Integrated Networks
JAR	Java Archive
JAVA EE	JAVA Platform, Enterprise Edition
JMX	Java Management Extensions
JNDI	Java Naming and Directory Interface
JSLEE	JAIN Service Logic Execution Environment
JAIN SLEE	JAIN Service Logic Execution Environment
JSR	Java Specification Request

L

LAN	Local Area Network
LESS	Language for End System Services
LSBB	Logic Service Building Block

M

MAMS	Multi-Access Modular-Services
MAP	Mobile Application Part
MEGACO	Media Gateway Control Protocol
MGW	Media Gateway
MGC	Media Gateway Controllers
MMS	Multimedia Messaging Service
MSBB	Management Service Building Block

N

NA	Network Abstraction
NGN	Next Generation Networks
NM	Normalized Message
NMR	Normalized Message Router

O

OASIS	Organization for the Advancement of Information Standards
ODSDP	Open Distributed Service Delivery Platform
OMA	Open Mobile Alliance
OMG	Object Management Group
OPUCE	Open Platform User-centric Service Creation and Execution
ORB	Object Request Broker
OSA	Open Service Access
OSE	OMS Service Environment
OSS	Operation Support System

P

P2P	Peer-to-Peer
PEEM	Policy Evaluation, Enforcement, and Management
PLMN	Public Land Mobile Network
POP	Point of Presence
PSTN	Public Switched Telephone Network

Q

QoS	Quality of Service

R

RA	Resource Adaptor
RCSBB	Resource Connection Service Building Block
RFC	Request for Comments
RPC	Remote Procedure Call

RTP	Real-time Transport Protocol

S

SBB	Service Building Block
SCE	Service Creation Environment
SCF	Service Capability Features
SCMSBB	Service Control Management Service Building Block
SCML	Service Control Mark-up Language
SCP	Service Control Point
SCS	Service Capability Server
SCXML	State Chart XML
SD	Service Deployment
SDP	Session Description Protocol
SEE	Service Execution Environment
SGW	Signalling Gateway
SIB	Service Independent Building Blocks
SIP	Session Initiation Protocol
SLEE	Service Logic Execution Environment
SMS	Short Message Service
SOA	Service-Oriented Architecture
SOAML	Service-Oriented Architecture Modelling Language
SPICE	Service Platform for Innovative Communication Environment
SPL	Service Processing Language
SQL	Structured Query Language
SS7	Signalling System No 7
STL	Service Transport Layer

T

TCP	Transmission Control Protocol
TLS	Transport Layer Security
TV	Television

U

UA	User Agent
UAC	User Agent Client
UAS	User Agent Server
UDDI	Universal Description and Discovery Interface
UDP	User Datagram Protocol
UML	Unified Modeling Language
UMTS	Universal Mobile Telecommunications System
URI	Uniform Resource Identifier
USSD	Unstructured Supplementary Services Data
UUS	User-to-User Signalling

V

VoIP	Voice over IP
Voice XML	Voice Extensible Mark-up Language

W

W3C	World Wide Web Consortium
WS	Web Service
WSDL	Web Service Description Language
WWI	Wireless World Initiative
WWW	World Wide Web

X

XCAP	XML Configuration Access Protocol
XML	Extensible Mark-up Language

Appendix B – Publications and Presentations

The following list includes publications and presentations related to the area of this research, to which the author of this thesis has contributed during the course of research.

1. Eichelmann, T.; Fuhrmann, W.; Trick, U.; Ghita, B. (2008), "Creation of value-added services in NGN with BPEL", Internal Publication, In (Bleimann, U.; Dowland, P.S; Furnell, S.M.; Grout, V.M.) *Proceedings of the Fourth Collaborative Research Symposium on Security, Elearning, Internet and Networking (SEIN 2008)*, University of Plymouth, School Of Computing, Communications And Electronics, UK, 2008, ISBN: 978-1841021966, pp186–193
2. Eichelmann, T.; Fuhrmann, W.; Trick, U.; Ghita, B. (2009), "Support of parallel BPEL activities for the TeamCom Service Creation Platform for Next Generation Networks", Internal Publication, In (Bleimann, U.; Dowland, P.S.; Furnell, S.M.; Grout, V.M.) *Proceedings of the Fifth Collaborative Research Symposium on Security, E-learning, Internet and Networking (SEIN 2009),* University of Plymouth, School Of Computing, Communications And Electronics, UK, 2009. ISBN: 978-1-84102-236-9, pp69–80
3. Eichelmann, T.; Fuhrmann, W.; Trick, U.; Ghita, B.V. (2010) "Enhanced Concept of the TeamCom SCE for Automated Generated Services Based on JSLEE*", Proceedings of the Eighth International Network Conference (INC 2010)*, In (Bleimann, U.; Dowland, P.S.; Furnell, S.M.; Schneider, O.) University of Plymouth, School Of Computing, Communications And Electronics, UK, 2010, ISBN: 978-1-84102-259-8, pp75-84
4. Eichelmann, T.; Fuhrmann, W.; Trick, U.; Ghita, B.V. (2011) "Discussion on a Framework and its Service Structures for generating JSLEE based Value-Added Services", *Proceedings of the Fourth International Conference on Internet Technologies & Applications (ITA 11)*, In (Grout, V.M.; Picking, R.;

Oram, D.; Cunningham, S.; Houlden, N.) North East Wales Institute, UK, 2011, ISBN: 978-0-946881-68-0, p 169-176

5. Eichelmann, T.; Fuhrmann, W.; Trick, U.; Ghita, B.V. (2012) "A JSLEE based Service Creation and Service Delivery Framework for value-added services in Next Generation Networks", *Proceedings of the 3rd International Conference on Internet and Applications 2012 (ITAP 2012)*, Wuhan, China, 2012, ISBN: 978-1-4577-1575-4
6. Lehmann, A." Eichelmann, T." Trick, U. (2008b), "Neue Möglichkeiten der Dienstebereitstellung durch Peer-to-Peer-Kommunikation" (translated title: "New ways of service provisioning through peer-to-peer communication"), *ITG-Fachbericht 208 Mobilfunk*, VDE-Verlag
7. Lehmann, A.; Eichelmann, T.; Trick, U.; Lasch, R.; Tönjes, R. (2009), "TeamCom: A Service Creation Platform for Next Generation Networks", In (Perry, M.; Sasaki, H.; Ehmann, M.; Bellot, G.O.; Dini, O.) *The Fourth International Conference on Internet and Web Applications and Services (ICIW 2009)*, Venice 2009, IEEE Computer Society, Los Alamitos, CA, USA, 2009, ISBN 978-0-7695-3613-2

Copies of the papers most closely related to the research described are enclosed within this appendix.

Published in *Proceedings of the Fourth Collaborative Research Symposium on Security, Elearning, Internet and Networking (SEIN 2008)* pp186–193, University of Plymouth, School Of Computing, Communications And Electronics, UK, 2008 ISBN: 978-1-84102-196-6

Creation of value added services in NGN with BPEL

T.Eichelmann[1, 2], W.Fuhrmann[3], U.Trick[1], B.Ghita[2]

[1] Research Group for Telecommunication Networks, University of Applied Sciences Frankfurt/M., Frankfurt/M., Germany
[2] Network Research Group, University of Plymouth, Plymouth, United Kingdom
[3] University of Applied Sciences Darmstadt, Darmstadt, Germany
e-mail : eichelmann@e-technik.org

Abstract

The telecommunication industry has recognized the potential of value added services. To be competitive, the companies have to react fast on market changes and on emerging trends. They constantly have to offer new services which are requested by the customers. Also smaller companies who maybe have nested in a niche have to adjust and extend there services. However for the implementation of value added services experts are needed and the implementation of a new service is very time consuming. Existing tools either are coarse-grained or do not offer modern modes of communication such as videoconferencing or web services required by value added services. In this paper a novel approach is presented, showing how services can be designed with the business process execution language (BPEL) and implemented with a service creation environment (SCE) within a short period of time, so that only BPEL-knowledge is required to build a new service. With this approach the service providers are able to offer custom-made services considering their customers requirements and conceivabilities.

Keywords

value added services, service creation, BPEL, JAIN SLEE, NGN

1. Introduction

In the past, telephony comprised simple audio connection between two participants. With the new emerging possibilities like videoconferencing, instant messaging, presence or web services value added services has broadening their scope. Service developers need new tools and development frameworks which can cope with the new requirements.

In the last years some frameworks and tools were developed, supporting the creation of value added services. However the problem of those tools and frameworks is that for the creation of value added services, specialists with extended knowledge are needed and the development process takes a lot of time.

For this reason description languages and graphical service creation environments were developed. The disadvantage of these description languages are their limited possibilities. Often it is only possible to combine prefabricated services into new more powerful services, resulting in a strong coarseness. Furthermore, today's services also require interfaces to other resources such as web services and databases. Also audio conferencing and video conferencing should be available.

This paper proposes a service creation environment that makes the following scenario possible. A customer has an idea for a service and formulates his service idea in textual form. His service provider analyses the service and designs it with the help of BPEL (described in chapter 2.1). Therefore the provider uses the building blocks (described in chapter 3) that are represented by the partner links within BPEL. The resulting BPEL process is not necessarily executable on a BPEL-engine. The service creation environment only uses the resulting xml-files from the BPEL process and transforms the BPEL files to java code that is deployable on an application server (described in chapter 2.2).

2. Used Technologies

2.1. BPEL

The business process execution language is a XML-based language to specify business processes. The activities of the business processes are implemented as web services. BPEL was defined by Microsoft, IBM and BEA in 2002 as BPEL4WS 1.0 (BEA *et al*., 2002). Version 2.0 was defined by OASIS as WS-BPEL 2.0 in 2004 (OASIS, 2004).

BPEL processes can invoke other web services and they can be invoked from other web services (Figure 1). These services are called partner. A partner communicates with the BPEL process through the web service. This web service interface is defined in the web service description language (WSDL). The simple object access protocol (SOAP) is used as transport protocol between the web services.

There are two ways BPEL processes can be described. One way is to describe an executable business process, which can be run on a BPEL process engine. The other way is the description of an abstract business process.

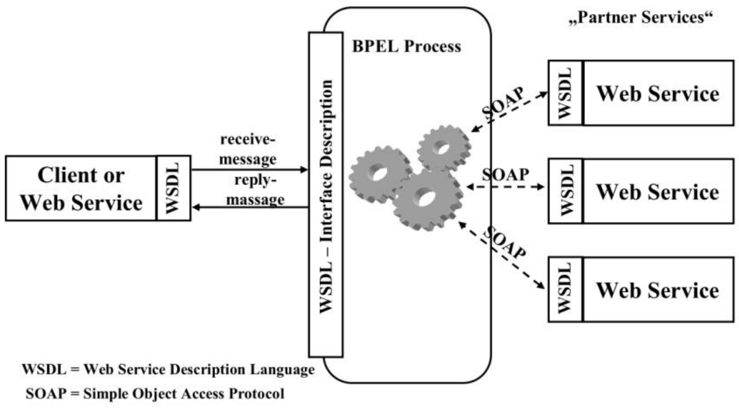


Figure 1: BPEL WSDL Interface

Abstract business processes have the objective to describe only the external visible aspects of the BPEL process. To interact with other web services the BPEL process has to describe the external behaviour which is relevant for the partners. In this approach abstract business processes are used to describe the communication building blocks as partners of the main BPEL process. These building blocks are implemented in java and are not required to be implemented as executable web services in BPEL.

2.2. JAIN SLEE

JSLEE or JAIN SLEE (Sun and Open Cloud, 2008) is the Java standard for SLEE (Service Logic Execution Environment). JAIN means Java APIs for intelligent networks. JSLEE is a high throughput, low latency event processing application environment for communication services. It allows the implementation of scalable high available communication services. Figure 2 shows an overview of the JAIN SLEE application server. The access to network resources is offered by resource adaptors (RA). The services are represented as service building blocks (SBBs). Invocations are transmitted asynchronously via events.

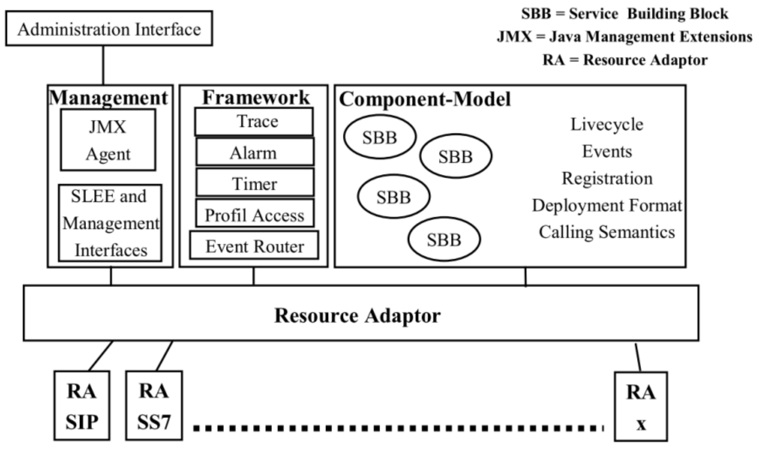


Figure 2: JSLEE Overview

3. Communication Building Blocks

This paper proposes communication building blocks which are made available for the service designer in BPEL as partners. These partners are connected with the process via partner links. Normally external resources such as web services are bound to these partner links.

The partners are communicating with the BPEL process over the WSDL interface. To allow for communication between the BPEL process and the building blocks a web service interface for the building blocks has to be defined. If the functionality of the building blocks increases, the corresponding WSDL interfaces must also be extended.

The building blocks categorize the functionalities provided by the Jain SLEE environment and its resource adapters. The goal was to find a small number of re-usable communication building blocks which cover all necessary functionalities. The functionalities are subdivided into eight communication building blocks: Audio, video, text, files, conference, data input, data output and data trigger. With the combination of some or all of these eight building blocks every value added service can be built.

4. The Service Creation Environment

4.1. Overview

To create custom-made value added services the customer formulates a workflow of the service idea and describes all requirements and conceivabilities for the value added service. His service provider analyses the service and designs a custom-made service with the help of BPEL.

The communication building blocks in BPEL are represented as partners. From these partners the designer can use the necessary methods by invoking them on the communication building blocks. The finished BPEL process is handed over to the service creation engine. There a parser examines the files and builds the service from it.

The service consists of the java classes that represent the JSLEE service building blocks and the corresponding descriptor-files. These descriptor-files are xml-files required for deploying the services and offering configuration options. Java classes, descriptor-files and other dependencies as for example libraries are all copied together into the same workspace. There a compiler creates the class files of the service. After compiling the service creation in completed and the service can be deployed on a JSLEE application server. The service creation process is shown in Figure 3.

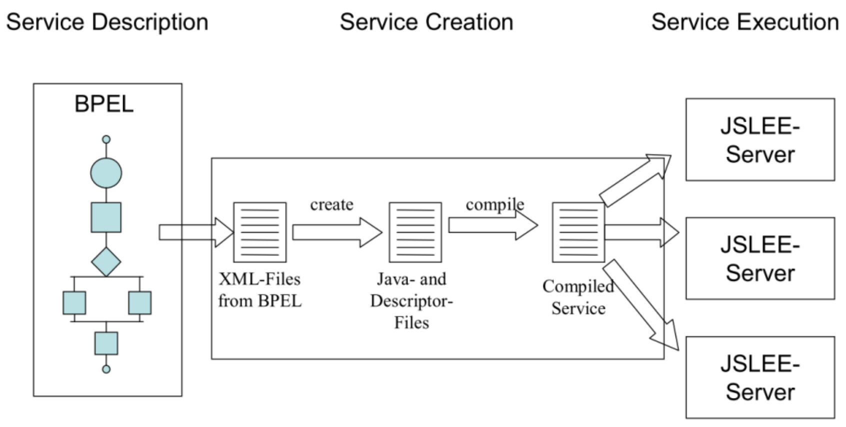


Figure 3: Overview of service creation

4.2. Service description in BPEL

The service is described in BPEL. The communication building blocks are represented in BPEL as partners, shown in Figure 4. The links between the BPEL process and its partners are described by partner links.

The partner links are described in WSDL files. Therefore, for every building block a WSDL description is needed to define the available methods and attributes, so that they become available in BPEL. The building blocks are abstract BPEL processes. The implementation of an executable BPEL process is not necessary, only the message exchange between partners is required to be defined.

Information from the building blocks arrives as events. Information for the building blocks is transported via method calls to the communication building blocks. Within the compiled service a building block calls the corresponding Resource Adaptor which sends out the protocol-specific information. The resulting BPEL process is not executable on a BPEL process engine. The BPEL process design tool produces as output the *.wsdl, *.xsd, and *.xml files. These files serve as input for the service creation.

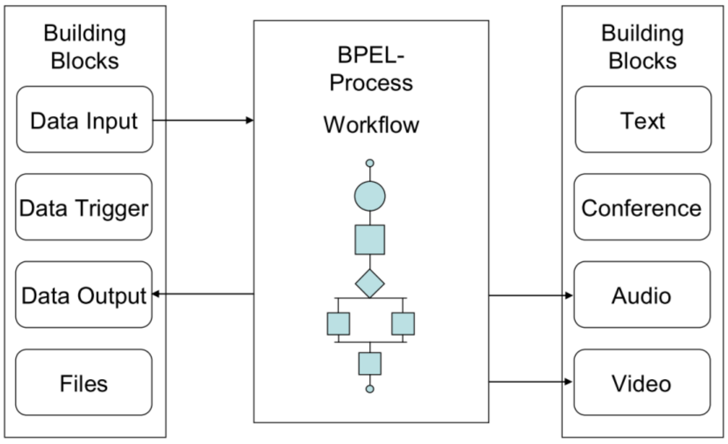


Figure 4: BPEL and the Communication Building Blocks.

4.3. Service Creation

The service creation will transform the BPEL process to java source code. The resulting files from the BPEL process design tool are serving as input. A parser will parse the BPEL files step by step. For every method called on a communication building block, the parser will add predefined code-snippets to the java code of the JSLEE SBBs and xml elements to the descriptor-files.

The descriptor-files describe the elements of the service, such as the SBBs, the events, and the RAs. The communication building blocks used in BPEL are defined as Java code for JSLEE. The java representation of the communication building blocks comprises the resource adaptor calls. Every building block is mapped to one or more resource adaptors. The BPEL activities and the service workflow are also mapped to java code snippets. For every element in BPEL a predefined java equivalent is required. These java snippets are structured in XML-Format. For every BPEL element a list of xml-java code snippets exists which is dedicated to these element.

When the parser reads the BPEL process, for every element that was found, the associated xml-java snippets are added to the source code. For every new resource adaptor to be added to the service creation these xml-java snippets have to be defined. Therefore the java code of the resource adaptor has to be transformed into the corresponding xml format. The new methods and attributes which are needed to access the RA from the building blocks must be added to the corresponding BPEL partner link.

The service creation will create a complete workspace with the needed java code, descriptor-files, build-files, and libraries. After the creation of the java files and the workspace the java compiler creates the service. Now the service can be deployed on a JSLEE application server. An overview of the service creation is shown in Figure 5.

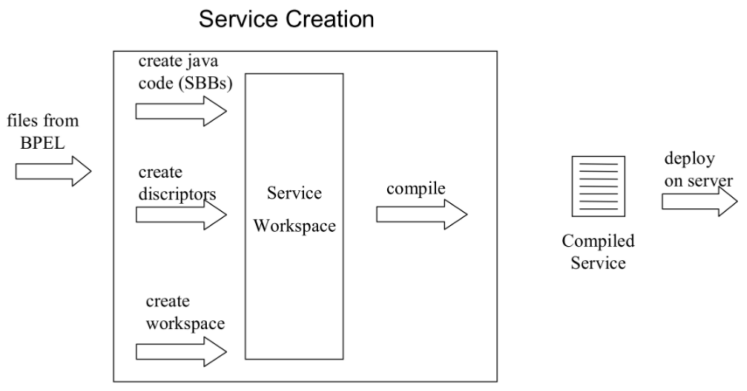


Figure 5: Service Creation

5. Comparison with other Approaches

5.1. Call Processing Language (CPL), Language for End System Services (LESS), Session Processing Language (SPL) and VisuCom

CPL (Rosenberg *et al*., 1999) is an xml-based language. It mimics the structure of IN service creation. Users can write a script or use a graphical design tool to develop a service. To make the service available, the user has to send it to his service provider.

LESS (Wu and Schulzrinne, 2003) is, like CPL, an xml-based language. It is easy to understand also for non-programmers and offers safety, simplicity, and extensibility.

It contains commands and events that provide direct interaction and control of media applications to users, media applications and other end system applications.

SPL (Burgy *et al*., 2006) is a domain-specific language which allows for the development of robust telephony services and offers an abstraction over the underlying protocols and software layers.

VisuCom (Latry *et al*., 2007) is a graphical telephony service creation and execution environment over SPL. It enables non-programmers to define services. It offers intuitive visual constructs and menus that permit users a quick development of telephony services.

However the scope of these approaches is limited to end-user services and the scripting languages are mostly coarse-grained. The approach defined in this paper allows a finer granularity and a wider range of communication modes. New resources can be added through JAIN SLEE by implementing new resource adapters. To enable access to the new resources, java code snippets have to be defined and the methods to control the new resources have to be added to the communication building blocks.

5.2. DiaGen

DiaGen introduces a declarative language over Java. Out of the declarative language, DiaGen (Jouve *et al*., 2008) generate a framework for the java programming language. This framework provides service discovery and high-level communication mechanisms. The framework generates the class skeletons of service classes. The programmer must fill in the service code.

However with the approach described in this paper a programmer has to write the java code only once. Then the resulting code snippets are mapped to the correspondent building blocks. New resources can be added through new resource adapters. These resource adapters can be developed by the service provider or acquired from the vendor of the JSLEE application server.

6. Conclusion and the Future

In this paper a new service creation environment was proposed. This work should overcome problems shown by other approaches. The goal of this SCE is a rapid and easy creation of customer-made value added services. The creation of new services can be achieved by service designer who understand the business process execution language.

The communication building blocks provide a simple, comfortable possibility for the designer to create the BPEL process. The SCE transforms the BPEL process to java code and builds the class-files. The JSLEE application server with its resource adapters offers defined interfaces which can mapped to the building blocks in BPEL.

The combination of JSLEE, BPEL, and the service creation engine allow a rapid development of value added services. The next step is the definition of building block methods with the corresponding java code snippets. A XML-structure has to be defined to structure the code snippets for code generation.

7. Annotation

The research project providing the basis for this publication was partially funded by the Federal Ministry of Education and Research (BMBF) of the Federal Republic of Germany under grant number 1704B07. The authors of this publication are in charge of its content.

8. Acknowledgment

I thank Armin Lehmann, Björn Ricks, Rolf Lasch and Patrick Wacht for the discussion and criticism of this work.

9. References

BEA Systems, IBM, Microsoft, Specification V1.0 (2002), “Business Process Execution Language for Web Services“, IBM developerWorks

Burgy L., Consel C., Latry F., Lawall J., Palix N., Réveillère L. (2006), “Language Technology for Internet-Telephony Service Creation”, *IEEE International Conference on Communications 2006*, Instanbul, ISBN: 1-4244-0355-3

Jouve W., Palix N., Consel C., Kadionik P. (2008), “A SIP-based Programming Framework for Advanced Telephony Applications”, *Principles, Systems and Applications of IP Telecommunications Services and Security for Next Generation Networks*, IPTCOM 2008 Heidelberg, Germany 2008

Latry F., Mercadal J., Consel C. (2007), ”Staging Telephony Service Creation: A Language Approach”, *In Principles, Systems and Applications of IP Telecommunications*, IPTComm, New-York, NY, USA, July 2007, ACM Press

OASIS (2004), OASIS Standard, “Web Services Business Process Execution Language Version 2.0”, OASIS

Rosenberg J., Lennox J., Schulzrinne H (1999), “Programming Internet Telephony Services”, *IEEE Internet Computing Magazine*, March 1999

Sun Microsystems, Open Cloud (2008), JSR-000240 Specification, Final Release, „JAIN SLEE (JSLEE) 1.1”, Sun

Wu X., Schulzrinne H. (2003), “Programmable end system services using SIP”, *IEEE International Conference on Communications*, May, 2003, 2nd New York Metro Area Networking Workshop, September 2002

Published in *Proceedings of the Fifth Collaborative Research Symposium on Security, E-learning, Internet and Networking (SEIN 2009)*, pp69–80, University of Plymouth, School Of Computing, Communications And Electronics, UK, 2009, ISBN: 978-1-84102-236-9

Support of parallel BPEL activities for the TeamCom Service Creation Platform for Next Generation Networks

T.Eichelmann[1, 2], W.Fuhrmann[3], U.Trick[1], B.Ghita[2]

[1] Research Group for Telecommunication Networks, University of Applied Sciences Frankfurt/M., Frankfurt/M., Germany
[2] Centre for Security, Communications and Network Research, University of Plymouth, Plymouth, United Kingdom
[3] University of Applied Sciences Darmstadt, Darmstadt, Germany
e-mail : eichelmann@e-technik.org

Abstract

The development of value added services is currently still very cost and time consuming. Companies have to react fast on market changes and on emerging trends. They constantly have to offer new services which are requested by the customers. The Service Creation Environment of the TeamCom project represents a promising approach. It supports the developer with the creation of services on the basis of re-usable service components called Communication Building Blocks and describes the service by a control logic. For the description of a value added service, a language, which was optimized for business processes, is suggested: the Businesses Process Execution Language (BPEL). The services described in BPEL are translated into Java code and compiled. For the deployment of the service, a Service Execution Environment, based on JSLEE is supported. This publication extends the TeamCom approach with the support of parallel processes. It is shown how forked BPEL activities can be translated into JSLEE services.

Keywords

SCE (Service Creation Environment); NGN (Next Generation Networks); BPEL (Business Process Execution Language); JAIN SLEE (JAIN Service Logic Execution Environment)

1. Introduction

The development of new services is currently still very time consuming and requires an extensive knowledge of the used technologies, as a consequence the penetration of specific multimedia services is low. So a high need of methods exists, which allow a simplification in the development of new value added services. Therefore the design of the services must be independent from the concrete implementation. This provides the advantage that the developer can concentrate on the actual service design, without the requirement of detailed knowledge of the used technologies. Here the TeamCom project (TeamCom, 2009) comes into play. It allows a fast, economical and efficient creation of value added services by providing a Service Creation Environment for an abstract design of the service process and an automatic creation and deployment of value added services. For the design of the services, the Business Process Execution Language (BPEL) (OASIS, 2004) is used, because it is an established business specification language. However, a BPEL engine is appropriate for Web Services but it is not suitable for the control of real time communication services. Therefore another approach is suggested, which converts the description of a business process into executable code. The generated code should be based on the JSLEE (Sun and Open Cloud, 2008) architecture, because this architecture is adapted on the requirements of communication services. This paper is an extension of the TeamCom approach. In the past, the TeamCom approach has only supported sequential services. This paper shows how forked BPEL activities can be translated into parallel running service components in JSLEE services. The Content of this paper is structured as follows: In chapter 2 the architecture of the service platform is described. The 3rd chapter presents the concept of the Service Creation Environment of the TeamCom project. Chapter 4 describes the lifecycle of the services from the idea, up to the realization. The conversion of forked BPEL processes to JSLEE services is described in the 5th chapter.

2. Architecture

The TeamCom project proposes an integrated architecture (Lehmann *et al*., 2009) (Lasch *et al*., 2009) (Lasch2 *et al*., 2009) for service creation, deployment and execution. This architecture (figure 1) can be divided into four layers: The Service Creation Environment (SCE), which includes the design and composition of new services, the Service Deployment (SD), the Service Execution Engine (SEE) containing one or more Application Servers (AS) based on JAIN SLEE (JAIN Service Logic Execution Environment) and finally the Service Transport Layer (STL). The Service Transport Layer abstracts different protocols in order to enable upper service layers to be independent of a specific communication protocol. Therefore it supports several communication networks, e.g. IP Multimedia Subsystem (IMS) (TS 23.228, 2006) or Peer-to-Peer SIP (P2PSIP, 2009).

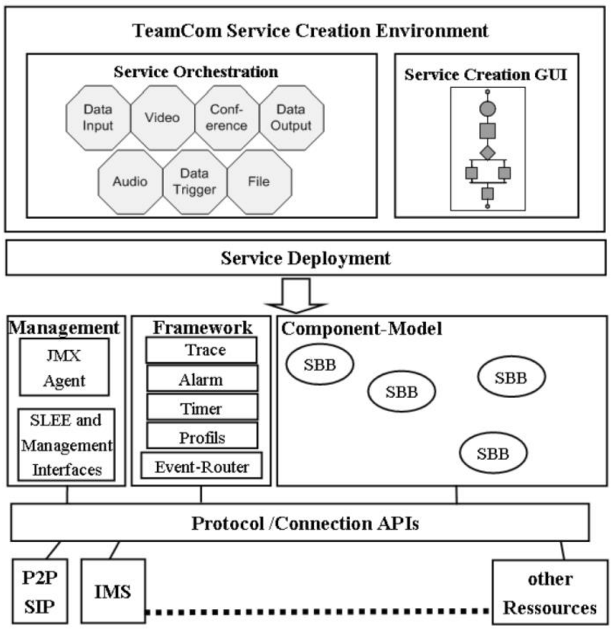


Figure 1: TeamCom Architecture

The SCE contains a GUI, permitting a developer to orchestrate service components and to integrate external services easily without having to develop low-level software code. The Service Execution Environment establishes a layer, where created Service Components are deployed and activated. In order to support NGNs that are based on IP the following requirements for a Service Execution Environment have been defined: independence from the operating system, service orchestration based on components, comfortable component deployment, support for SIP (Rosenberg *et al*., 2002), extensibility for other protocols and possibility for co-operation between numerous application servers. To fulfil these requirements the architecture makes use of the JAIN SLEE standard. The JAIN SLEE standard defines a component, event and transaction based architecture. The architecture is written in the Java language and standardised by a Java Community Process. It is part of the JAIN (Java API for Integrated Networks) Initiative consisting of several telecommunication companies. JAIN SLEE is designed for ensuring low latency and providing high throughput to accomplish the requirements for communication services. It uses a distributed component model similar to Enterprise Java Beans (EJB) (Sun and Oracle, 2006). In the standard so called Resource Adaptors (RA) are defined abstracting the underlying infrastructure. These Resource Adaptors provide a common Java API which hides the communication protocol underneath. In detail when a communication protocol message is received the corresponding RA translates this message into a Java event class. Afterwards the event and an activity class both together are passed to the JAIN SLEE event router. A JAIN SLEE activity represents a session of a communication protocol e.g. a SIP dialog. The event router utilizes these objects to look up the services which requested to receive the specific event. Accordingly the service itself is able to react on the event and to create an answer by using defined Java Interfaces. The answer is translated to a communication protocol response by a RA. The service itself is composed of one or more Service Building Blocks (SBB). These SBBs contain the application/service execution logic and are deployed on a JAIN SLEE Application Server such as Mobicents (Mobicents, 2009). The SBB component model includes a lifecycle, registration and security management. In addition, SBBs are able to access timer, trace, alarm and profile facilities which are also provided by a JAIN SLEE server.

3. Service Creation Environment

As depicted in the last chapter the Service Creation Environment includes service orchestration on the basis of reusable Service Components and existing services. In addition to the components a service logic is required for describing the workflow properly. The following two sections explain both.

3.1. Communication Building Blocks

Eight elementary Service Components called Communication Building Blocks are derived from the requirements for telecommunication services. By combining these elementary components it should be possible to describe and generate other value added services. These Communication Building Blocks comprise audio, video, text, file, conference, data input, data output and data trigger components. This chapter discusses the abstract Service Components in detail.

Audio: The Audio Component handles all kind of audio communication including the establishment of a call, answering calls, manipulation of audio streams (e.g. mixing, transcoding) and sending and receiving DTMF tones.
Video: The Video Component is responsible for playing and recording of video streams. It enables to create and close video calls and to combine different video signals for merging a new video stream.
Text: This component exchanges messages between two partners and has the capabilities of handling strings, e.g. search for a specific word in a text, replace alphabetic characters or change the encoding of a text.
File: The File Component handles creation, deletion, sending and receiving of binary files. Another task of File is to write and read any kind of data from and to any position in a file. Finally this component is able to rename files or directories.
Data Input: All kind of data queries are processed by the Data Input component. This includes database queries as well as reading data from a sensor.
Data Output: The counterpart of Data Input is Data Output being concerned with writing data to a destination e.g. to a database or to control an actuator.
Conference: This is a special kind of communication component because it re-uses internal functions of the previously described components, e.g. audio stream mixing.
On top of this, the Conference Component provides functionalities for creating and deleting conference "rooms", adding and removing users to/from a room.
Data Trigger: The Data Trigger is closely related to an event generator. If a specific data trespasses a value an event is triggered. This data can be a sensor value, a timestamp or periodical dates.

3.2. Service Logic

As service logic for describing the workflow, BPEL (Business Process Execution Language) is used. BPEL is an XML (W3C, 2008) based process description language, building completely on web services. Most often BPEL is used to orchestrate web services but the language is protocol independent. It is possible to create so called bindings which could specify the usage of BPEL for other protocols than SOAP (Simple Object Access Protocol) (SOAP, 2007). In BPEL it is possible to define synchronous and asynchronous processes. Thus it fits very well to be used in combination with JAIN SLEE. For a structured process different BPEL tags are defined, e.g. sequence, if, while. Moreover forked processes can be created, allowing for execution in parallel and synchronisation afterwards. A BPEL process has the ability to invoke other services asynchronously and itself can be invoked from other services.

4. Service Creation and Deployment

Service creation and deployment (Eichelmann *et al.*, 2008) can be realised in five steps: writing a non-technical description of the service, converting this description to a service description language, analysing the description language, generating Service Building Blocks from the description language and deploying the service. First a service description has to be verbalised. The description could be e.g. a text, depending on the process and the involved people. Afterwards a service developer is able to create a BPEL based description graphically via the service creation Graphical User Interface (GUI). In this step the service developer utilizes the

Communication Building Blocks which are included in BPEL as partner links and configures their input. After finishing the BPEL process description the generation of the service will start (see Figure 2). The service designed by the service developer shall not be executed on a BPEL process engine. Instead the service shall run on a JAIN SLEE application server.

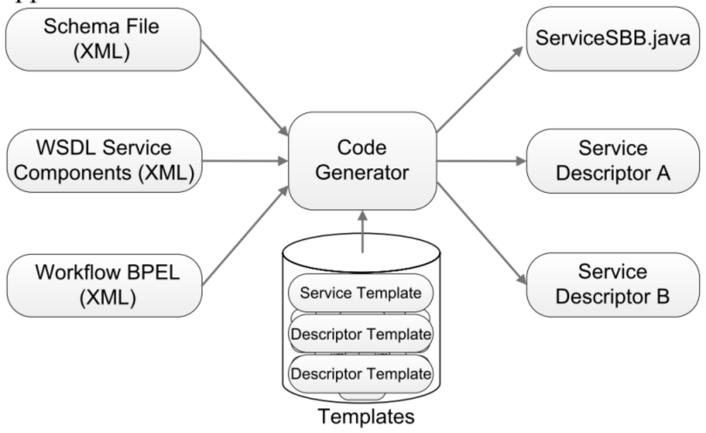


Figure 2: CodeGenerator

So the BPEL process has to be converted in a form to be executable as a JAIN SLEE service. The code generator analyses the BPEL process and parses the workflow step by step. The result from each individual step is saved in template files. Finally the goal of the code generator is the creation of the Java classes and the necessary descriptor files needed for a JAIN SLEE service. While the code generator parses the BPEL process, it analyses the BPEL activities. Pending on the BPEL activity the code generator adds pre-defined Java fragments to the template files. Process activities which initiate events for the partners or waiting for events from them must be examined to figure out which Communication Building Block is affected by this event. For each method which can be invoked on a partner a pre defined Java method exists. If the Communication Building Block is identified, the appropriate Java method which represents the used method from the BPEL process can be inserted into the template file. Other workflow activities perhaps need variables or data structures which are defined in BPEL. These structures are represented in XML schemata and have to be transformed into Java code also.

5. The conversion from BPEL processes to JSLEE Service Building Blocks

The BPEL process has to be converted to a JAIN SLEE service. This conversion is done by the Code Generator, which has to decide, whether the actual activity can be handled sequentially or if this activity requires parallel processing.

5.1. Conversion of sequential BPEL activities to SBBs

Sequential BPEL processes can be translated in JSLEE into one single SBB. In figure 3, a simple BPEL process is shown, which contains only three activities within its main sequence, a receive activity called receiveInput, an assign activity called assign, and an invoke activity called invokeCallback.

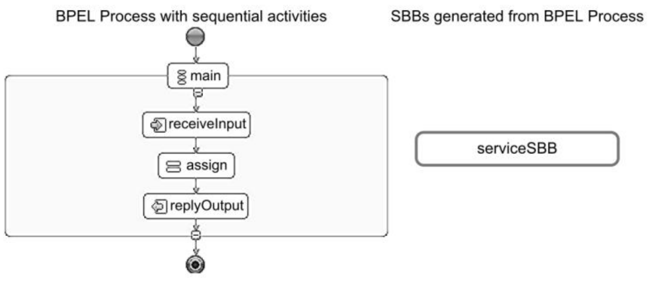

BPEL Process with sequential activities
SBBs generated from BPEL Process
main
receiveInput
assign
replyOutput
serviceSBB

Figure 3: BPEL process with sequential activities and the generated Sbb

The receive activity of the process waits for an arriving event. After the event has arrived, the content of the variables can be manipulated by the assign activity. If this process for example represents an echo service, which returns an arriving text back to the sender, the assign copies the arriving text from the input string variable into the output string variable. If the text should get send back to the sender, the destination address of the message receiver can also be set by the assign activity. If the message is ready, it can be sent by the invoke activity. This BPEL process contains only sequential BPEL activities, which means, that only one SBB is necessary for the JSLEE service.

5.2. Conversion of parallel BPEL activities to SBBs

If the Code Generator is reaching a flow activity while parsing a BPEL process, it is generating a new BPEL process from each branch it finds within the flow sequence. Figure 4 shows newly generated BPEL processes from the flow branches of a flow activity within the main sequence of a simple process.

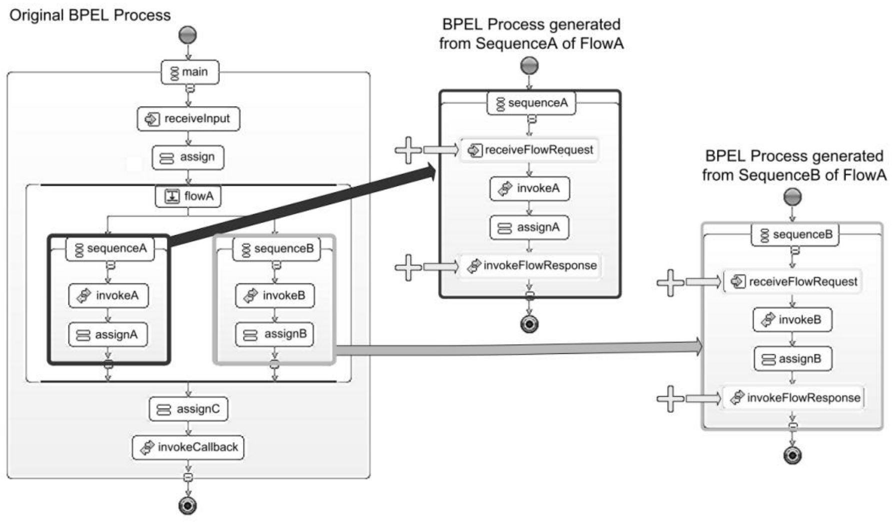


Figure 4: BPEL processes generated from a flow activity

This flow activity contains two flow branches. From every branch a new BPEL process will be generated. Every new process additionally gets a receive activity as first activity of the BPEL process and an invoke activity as last activity of the process. Later, from these two activities the SBB Java methods will be generated which sends and receives events. The newly generated BPEL processes are handed over to the Code Generator again. Now the Code Generator can generate new SBBs from the BPEL processes. If there are multiple flow activities nested within a BPEL process, this is repeated until all flow branches are transformed into new BPEL processes and are finally translated into SBBs. A forked JSLEE service component is required for the transformation of the BPEL flow activity. In order to use parallel running service components in JSLEE, the SLEE standard offers the possibility to use several SBBs in one service. These SBBs run in parallel from each other. Each SBB can either form its own service or several SBBs can form a single service together. In this work, several SBBs are used to represent the flow activity. Each branch of the flow activity is represented in its own SBB. If a service consists of several flow activities, it is distinguished if they are contained in the same sequence (Section 5.3.1) or if the flow activities are contained within another flow activity (Section 5.3.2). Events are used to communicate between the SBBs. With these events, the necessary variables are transferred between the SBBs. Figure 5 shows a BPEL process, which was extended by a flow activity and an assign activity compared to the process in figure 3. The additional assign activity, which is called assignC is used to copy the returned variables from the flow SBB into the variables of the answer message.

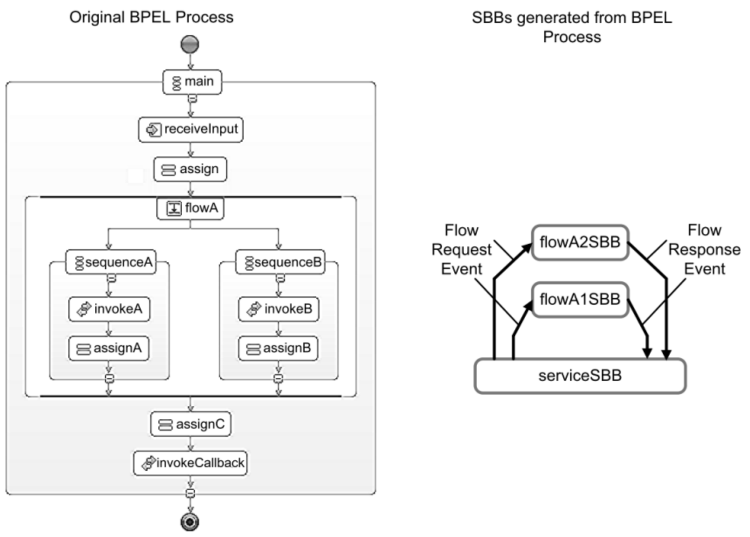


Figure 5: BPEL process with a single flow activity and the resulting SBBs

The sequences within the branches of the flow activity must be processed in parallel. If a sequence in a flow branch contains further sequential activities, those activities are sequentially processed within the SBB of the flow branch like the main sequence in the general SBB. The BPEL process in figure 5 contains two branches in its flow activity. The flow activity is called flowA and the sequences are called sequenceA and sequenceB. In this example both sequences contain two activities each, an invoke and an assign. Figure 5 shows the resulting SBBs from this BPEL process on the right side. The general SBB of this service called serviceSBB contains the sequential activities of the main sequence. In the SBB called flowA1SBB, the activities of the first flow branch are contained and in the SBB flowA2SBB, those of the second. The name of a SBB which is generated from a flow branch consists of the name of the flow and an index and the word SBB. Events are used for the communication between SBBs. Request and response events have been defined for the call and for the answer of SBBs. Each SBB which was generated from a flow branch is called with its own request event and answers with a response event after its processing. Both, the request and the response event contain the necessary variables which will become available in both SBBs in this way. The generated SBBs and the direction of the request and response events are shown on the right side of figure 5. A BPEL process is not only limited to one flow activity, it can contain as much flow activities as desired in different nesting levels. Two different nesting possibilities must be considered on the generation of the SBBs. On the one hand several flows can be contained in the same sequence; on the other hand further flow activities can also be contained in the branches of a flow.

5.3.1. Conversion of several flow activities within a sequence

Figure 6 shows a simplified BPEL process which contains two flow activities in the same sequence. The first flow activity is called flowA and the second is called flowB. Each flow activity consists of two flow branches and in each branch a sequence with further BPEL activities is contained.

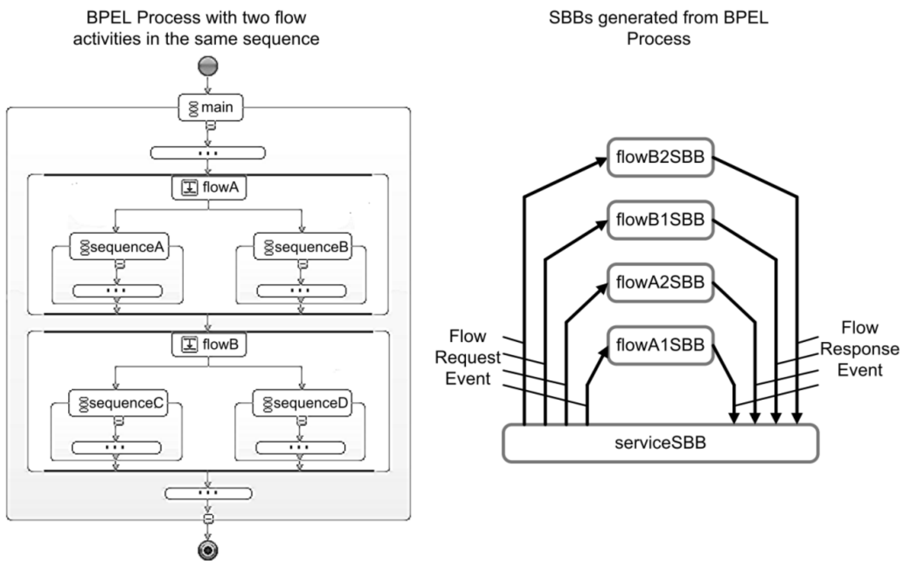


Figure 6: BPEL process with two flow activities within a sequence and the resulting SBBs

For each branch in each flow an SBB is generated. Each SBB represents the activities of the appropriate branch in Java code. So for example the flowA activity with the sequence sequenceA from the BPEL process shown in figure 6 is translated into an SBB with the name flowA1SBB. From this BPEL process the general SBB and four SBBs from the flow branches are generated. During the processing of the Java code representing the main sequence of the BPEL process, the flowA activity is reached first and request events are sent to activate the SBBs FlowA1SBB and FlowA2SBB. Both SBBs now process their tasks simultaneously. The general SBB serviceSBB is activated again if both SBBs reply with a response event. If both responses are received, the program code for the second flow flowB can be reached. Here the SBBs FlowB1SBB and FlowB2SBB are called with request events and they answer with response events. Afterwards the java code that represents the remaining activities of the main sequence can be processed.

5.3.2. Conversion of flow activities nested within a flow activity

As already mentioned above, flow activities can be nested within flow activities in BPEL. Figure 7 shows such a BPEL process.

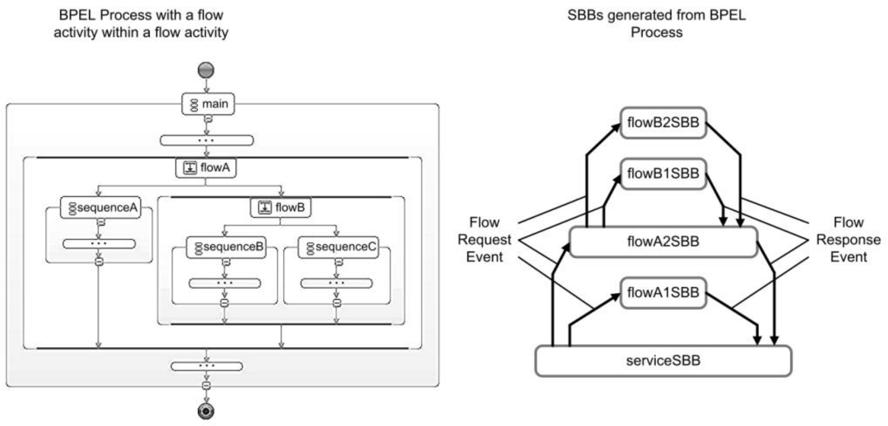


Figure 7: BPEL process with a flow activity nested within another flow activity and the resulting SBBs

FlowB is embedded in the right flow branch of flowA. An SBB is also generated for the flow branch of flowA that contains flowB. In this example the SBB is named flowA2SBB. The difference to flow activities within the same sequence is that the request events are sent by the SBB, in which the called flow is contained. In the example shown in Figure 7 the SBBs flowB1SBB and flowB2SBB are called by the SBB flowA2SBB, so the flowA2SBB sends the request events and also expects the response events. Therefore flowA2SBB can finish its processing only if flowB1SBB and flowB2SBB send their response events and the serviceSBB can continue with the processing if flowA1SBB and flowA2SBB are finished.

6. Conclusion

The automated service development has a high potential, it also empowers small and medium sized enterprises to develop their own value added services in a time and cost efficient way. The service development should be just as simple as the production of web pages and therefore be applicable for a broader range of people. The ease of service creation will foster a better degree of utilisation of the telecommunication's multimedia networks. The combination of the BPEL and JSLEE technologies simplifies the development of these value added services. In addition the abstraction from specific communication protocols, the re-usable Communication Building Blocks and the use of existing partial services facilitates the service development. The TeamCom architecture shows a continuous solution from the service development to the deployment and remains independent from specific protocols. The presented approach for the conversion of parallel BPEL activities into JSLEE SBBs additionally extends the TeamCom project with various possibilities of value added services. It also enables the offer of one or more SBBs for every BPEL activity. This extended approach could represent the basis for a BPEL engine on a JSLEE application server.

7. References

Eichelmann, T., Fuhrmann, W., Trick, U. and Ghita, B. (2009), “Creation of value added services in NGN with BPEL”, *SEIN*, Wrexham, 2008.

Gudgin M. (2007), "SOAP Version 1.2 Part 1. Messaging Framework (Second Edition)", W3C, April 2007.

Lasch, R., Ricks, B. and Tönjes, R. (2009), “Service Creation Environment for Business-to-business Services”, *SOCNE*, 2009.

Lasch, R., Ricks, B. and Tönjes, R. (2009), “Konzept eines BPEL zu JSLEE Compilers auf Basis wieder-verwendbarer Kommunikationsbausteine”, *Mobilkommtagung*, 2009.

Lehmann, A., Eichelmann, T., Trick, U., Lasch, R., Ricks, B. and Tönjes, R. (2009), “TeamCom: A Service Creation Platform for Next Generation Networks”, *ICIW*, Venice, 2009.

Mobicents Open Source JAIN SLEE Server Project Web Site (2009): http://www.mobicents.org. (Accessed 18 August 2009)

OASIS (2004), OASIS Standard, “Web Services Business Process Execution Language Version 2.0”, OASIS.

P2PSIP IETF Project Web Site (2009), http://www.p2psip.org/. (Accessed 18 August 2009)

Rosenberg J. (2002), RFC 3261 “SIP: Session Initiation Protocol” IETF, June 2002.

Sun Microsystems, Open Cloud (2008), JSR-000240 Specification, Final Release, „JAIN SLEE (JSLEE) 1.1”, Sun.

Sun Microsystems, Oracle Corporation (2006), JSR-000220 Specification, Final Release, "Enterprise JavaBeans, Version 3.0", SUN, May 2006.

TeamCom Project Web Site (2009): http://www.ecs.fh-osnabrueck.de/teamcom.html. (Accessed 18 August 2009)

TS 23.228 (2006), "IP Multimedia Subsystem (IMS); Stage 2 (Release 5)", 3GPP, June 2006.

W3C (2008), "Extensible Markup Language (XML) 1.0 (Fith Edition)", *W3C Recommendation*, November 2008.

Acknowledgment

The research project providing the basis for this publication was partially funded by the Federal Ministry of Education and Research (BMBF) of the Federal Republic of Germany under grant number 1704B07. The authors of this publication are in charge of its content.

Published in *Proceedings of the eighth International Network Conference (INC 2010)*, pp75-84, University of Plymouth, School Of Computing, Communications And Electronics, UK, 2010, ISBN: 978-1-84102-259-8

Enhanced Concept of the TeamCom SCE for Automated Generated Services Based on JSLEE

Thomas Eichelmann[1, 2], Woldemar Fuhrmann[3], Ulrich Trick[1], Bogdan Ghita[2]

[1] Research Group for Telecommunication Networks,
University of Applied Sciences Frankfurt am Main
[2] Centre for Security, Communications and Network Research, University of Plymouth
[3] University of Applied Sciences Darmstadt
eichelmann@e-technik.org

Abstract: The development of value added services is currently still very time and cost consuming. The TeamCom project offers a solution to cope with this problem. It offers a simple, fast, and cost efficient way to design the service graphically in a BPEL developer tool. A code generator analyses the BPEL code and generates the value added service. This service can be deployed on a JSLEE application server. The TeamCom project proved the possibility to generate value added services automatically. Nevertheless there are still some problems within the TeamCom approach and not all of them could be solved. This paper proposes an enhanced concept for the generation of value added services. This new concept cope with the problems from the old concept and it achieves a better integration into JSLEE.

1 Introduction

The market situation in the telecommunication sector forces the telecommunication industry to expand their business in the area of value added services. But the development of these services is very cost and time intensive. Furthermore the developers need a lot of detailed knowledge about communication systems and their protocols. The TeamCom [Te10] project offers a solution to solve this problem. It offers a simple, fast and cost efficient possibility for the developer to design the services with the help of a graphical user interface. A BPEL (Business Process Execution Language) development tool is used as graphical service design tool. The BPEL processes designed with this tool are analysed by a code generator and translated into the service code. Subsequently, the service can be deployed on an Application Server. JSLEE (Java Service Logic Execution Environment) [Su08] is used as service execution environment for the generated service. The TeamCom approach already proved its promises. Nevertheless some problems occur with the service structure of the present concept. The translation of the control structures from BPEL into the JSLEE service structure, require fundamental changes to the original TeamCom concept. Services generated with the code generator have a monolithic structure (chapter 3.2). With this monolithic structure the service generation was in some cases insufficient. The development of services that require parallel program execution was not possible with this structure. Multiple service components were required for the development of such services.

Many features that are offered by the JSLEE framework remained unused in the original concept. This paper proposes a new concept that was derived from the TeamCom approach. It avoids the problems of the old concept and offers a better integration into JSLEE. Furthermore the new concept offers a simplified expandability and a far better modularity.

The basic concept of the TeamCom project is illustrated in figure 1. It can be described with the following steps: writing a non-technical description of the service, converting this description to a formal service description language, analysing the formal description, generating a Service from the formal description and deploying the service [EFTG08].

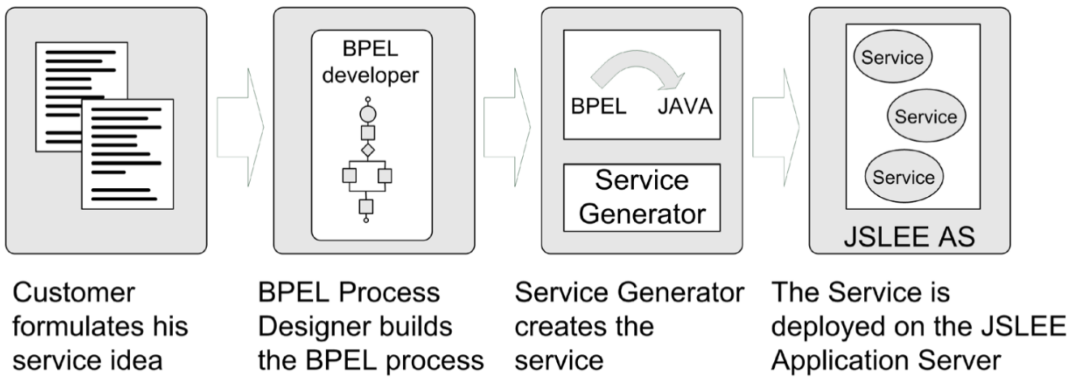

BPEL developer
BPEL JAVA
Service Generator
Service
Service
Service
JSLEE AS
Customer formulates his service idea
BPEL Process Designer builds the BPEL process
Service Generator creates the service
The Service is deployed on the JSLEE Application Server

Figure 1: TeamCom service development

This paper offers a new idea for the structure of the services, generated by the code generator. The generated service structure is derived from the designed BPEL process. For every process activity in BPEL the code generator creates a service component in JSLEE. The service components are communicating together via events.

2 Architecture

The architecture is designed to provide possibilities for service creation, deployment and execution [LET+09, LRT09a, LRT09b]. It is divided into four layers (figure 2), which includes the Service Creation Environment (SCE), the Service Deployment (SD) and the Service Execution Environment (SEE) containing one or more Application Servers (AS) based on JSLEE and finally the Service Transport Layer (STL). The Service Transport Layer abstracts different protocols in order to enable upper service layers to be independent of a specific communication protocol. Therefore it supports several communication networks, e.g. IP Multimedia Subsystem (IMS) [TS06].

As Service Execution Environment the JSLEE framework is used. JSLEE is designed for ensuring low latency and providing high throughput to accomplish the requirements for communication services. In the standard [JSLEE08] so-called Resource Adaptors (RA) are defined abstracting the underlying infrastructure.

These Resource Adaptors provide a common Java API that hides the communication protocol underneath. In detail when a communication protocol message is received the corresponding RA translates this message into a Java event class. Afterwards the event is passed to the JSLEE event router. The event router looks up the services that are interested in the specific event and delegate the event to these services. Accordingly the service itself is able to react on the event and to create an answer by using defined Java Interfaces. The answer is translated to a communication protocol response by a RA. The service itself is composed of one or more Service Building Blocks (SBB). These SBBs contain the application/service execution logic and are deployed on a JSLEE Application Server.

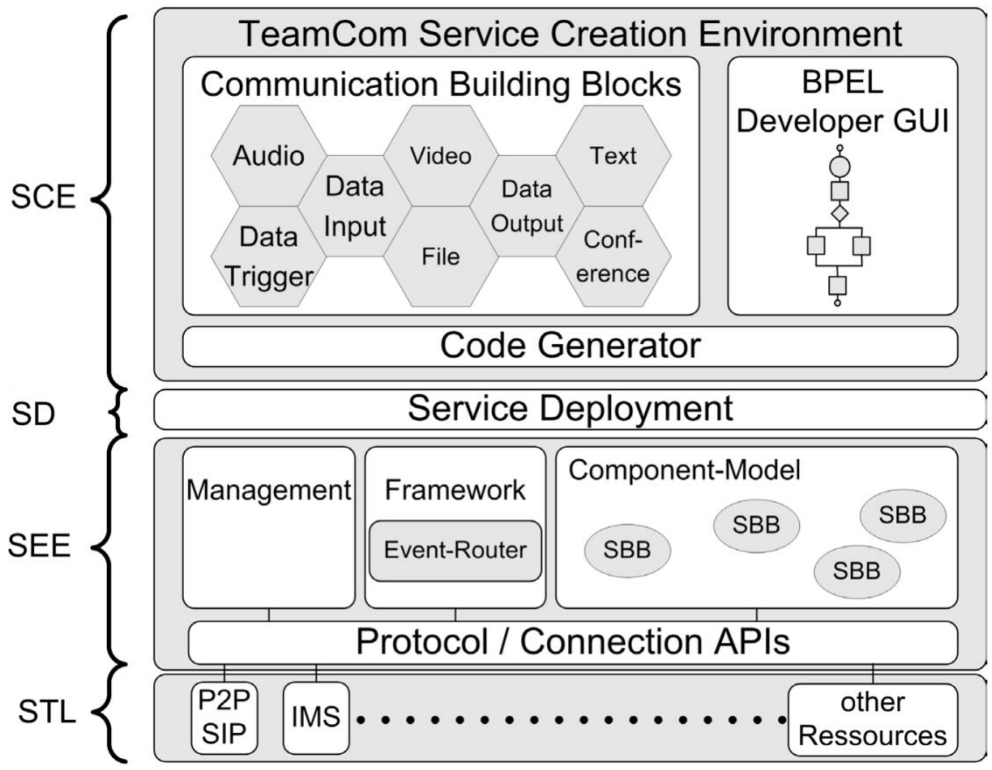


Figure 2: TeamCom Architecture

2.1 BPEL Developer GUI

The intention of the project is to generate services in a simple and fast way. Therefore the project requires a description language that is simple but powerful enough to describe telecommunication services. BPEL [OA04] was chosen as this language. BPEL is an established business process specification language normally used related to web services. It is based on XML (Extensible Markup Language) [W3C08] and it allows the service developer to use existing graphical BPEL design tools to design the service logic. The designed BPEL process does not need to be deployed on a BPEL engine. The process is used by the service creation environment to generate the telecommunication service.

2.2 Communication Building Blocks

Eight elementary Building Blocks called CBB (Communication Building Blocks) are derived from the requirements for telecommunication services. Services can be created by combining these CBBs. In the BPEL developer tool the required functionality can be invoked through partner links. For every CBB one partner link is available.

To make use of the functionality of a CBB in BPEL the corresponding method from the partner link that offers the required functionality has to be invoked. The code generator adds predefined java methods for every partner link within the BPEL process into the service code. These methods can be called from the SBB. The BPEL developer only requires the WSDL (Web Service Description Language) [W3C07] files to use the "virtual" partner link in the BPEL development tool. Figure 3 shows the representation of CBBs in BPEL and Java. On the left side the CBBs are represented as partner links.

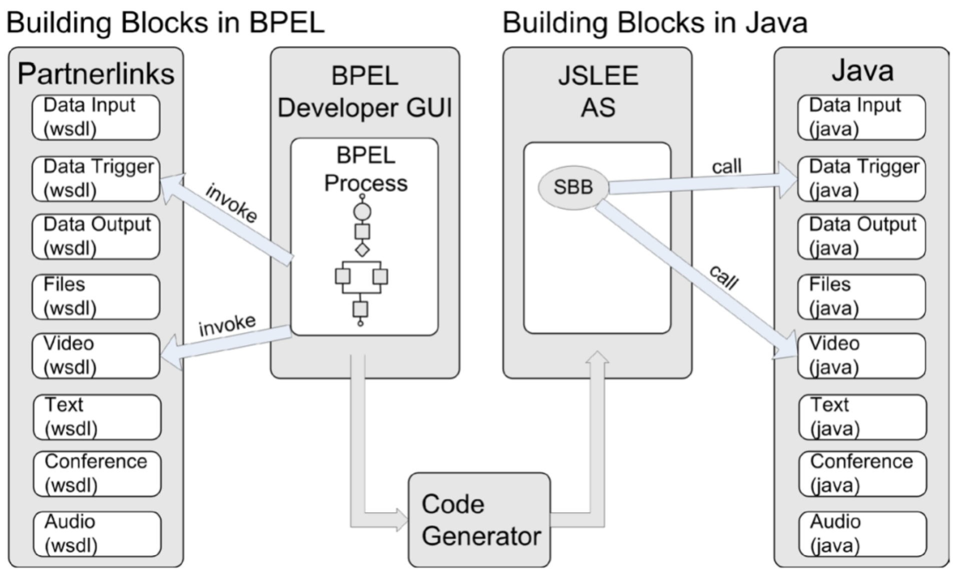


Figure 3: Representation of Communication Building Blocks

The methods from the CBBs are invoked from the BPEL process. On the right side the CBBs are represented as java methods that can be invoked from the generated SBBs.

2.3 Code Generator

As result from the service description part, the BPEL developer designs a BPEL process. In the next step, this BPEL process has to be analysed and a telecommunication service has to be generated with these information's. Here the code generator comes into play. The code generator analyses the BPEL process and parses the workflow step by step. The result from each individual step is saved in template files. Finally the goal of the code generator is the creation of the Java classes and the necessary descriptor files needed for a JSLEE service. While the code generator parses the BPEL process, it analyses the BPEL activities.

Pending on the BPEL activity the code generator adds pre-defined Java fragments to the template files. Process activities that initiate events for the partners or waiting for events from them must be examined to figure out, which Communication Building Block is affected by this event. For each method, which is defined within a partner link, a pre defined Java method exists.

If the Communication Building Block is identified, the appropriate Java method, which represents the used method from the BPEL process, can be inserted into the template file. Other workflow activities may need variables or data structures that are defined in BPEL. These structures are represented in XML schemata and have to be transformed into Java code also.

3 The generated Service in the TeamCom approach

In the TeamCom approach the Code Generator parses the BPEL process and generates a monolithic SBB. Only in the case that the BPEL process contains one or more flow elements, more then one SBB is generated. A flow is an element with multiple parallel paths. These paths can be executed in parallel.

3.1 Single SBB

A BPEL process without flow activities is translated into one single SBB (figure 4). Within this SBB a state machine controls the service workflow. The state machine decides about the events that are allowed to be received on the individual states. The state machine is generated by the Code Generator and represents the workflow of the BPEL process. It guarantees that the SBB is only allowed to listen on an event in a specific state.

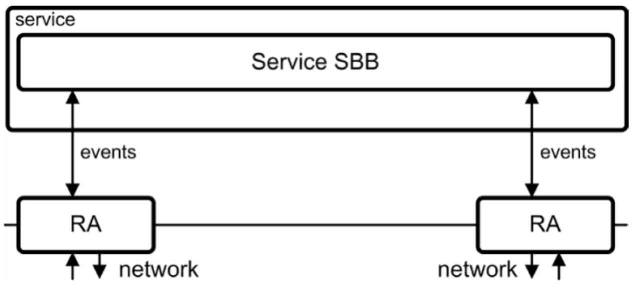


Figure 4: Monolithic TeamCom approach

3.2 Parallel program flow

Services require the possibility of parallel program execution. In BPEL the flow activity exists to describe parallel program execution. JSLEE only supports sequential program execution in one SBB. It is not allowed to use multithreading within an SBB. A possibility to use parallel program execution in JSLEE is to use more than one SBB.

They can be executed independent and in parallel from each other. The concept to use this characteristic to obtain parallel executed service parts is described in [EFTG09]. During compiling time, the code generator parses the BPEL process and analysis the activities. If a flow activity is detected, a SBB is generated from each branch within the flow. One SBB represents all activities from one branch of the flow activity. The generated SBBs (figure 5) are communicating with the help of events.

The SBB generated from the main BPEL process uses request events to signalise the SBBs that are generated from the flow activity to start processing.

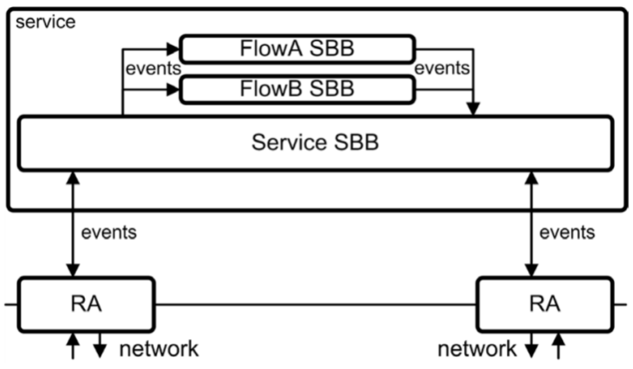


Figure 5: Monolithic TeamCom approach with flow

After the main SBB has fired his events to all flow SBBs, it waits for the returning response events. With these events the flow SBB receives the required parameter values from the main SBB. These values are used to initialise and to activate the SBBs. After processing of the SBB the changed parameter values are assigned to the response event and delivered back to the main SBB. After the main SBB has received the response events from all flow branches, the main SBB can copy the parameter values from the flow SBBs and continue processing

4 Representing BPEL activities as SBBs in JSLEE

In the previous chapter the TeamCom approach was declared. In this approach a monolithic SBB will be generated in most cases. Only for the flow activity, new SBBs are generated, which represent the flow branches of the BPEL process.

This chapter expands the idea of the flow SBBs. An SBB will be spent for each activity of the BPEL process. I.e., for every activity which exists in BPEL, a SBB is generated which represents the BPEL activities in JSLEE. These SBBs are called activity SBBs. In this paper two concepts are proposed. The first concept requires a special control SBB, which holds the state machine, all parameters and activities. The control SBB communicates between the activity SBBs. In the second concept, the generated SBBs control themselves. No special control SBB is needed. These self-controlled SBBs communicate directly with each other.

4.1 Control SBB concept

The service architecture which uses controlled SBBs requires an extra service component. A special control SBB is needed to control the service workflow and to coordinate the SBBs of the service. The control SBB assigns the work to the activity SBBs, sets the required parameters, and decides, on which events an SBB has to listen and what events he has to fire.

The control SBB starts on a service start event and initiates the required SBBs. The control SBB uses a state machine to decide which SBB should be called next. The internal state machine was generated from the code generator and derived from the BPEL process. In figure 6 the control SBB is activated by the service start event. The control SBB reads the info, which SBB is the next, from the state machine and fires an event to this SBB with the required parameter and the used CBB.

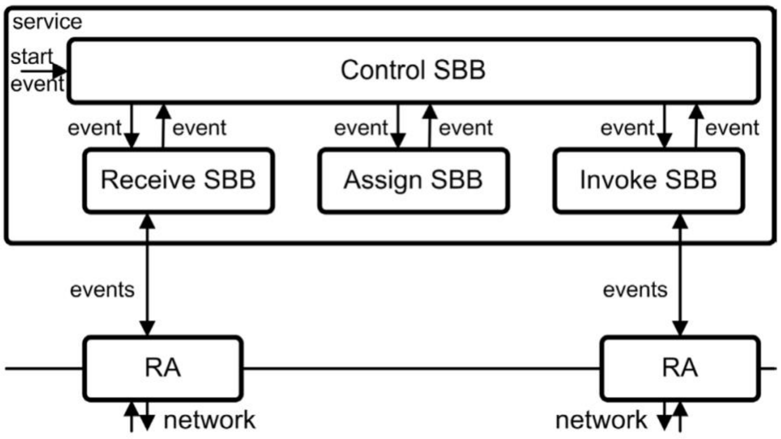


Figure 6: Service with controlled SBB architecture

The called SBB (in this case a Receive SBB) performs its work, e.g. communicating with a resource adaptor or calling methods from the CBBs. Afterwards the result is returned to the control SBB. With this information, the control SBB calls the next SBB according to the state machine. This procedure is repeated, until the workflow is completed.

4.2 Self controlled activity SBBs

Self controlled SBBs are not managed by a control SBB. They start working when they receive an event from its predecessor. The event includes all required parameters. So the SBB can start working after receiving the required event. The tasks the SBB has to perform are predefined and implemented in the SBB by the code generator. The methods the SBB has to call from the Java implementation of the CBBs (see chapter 2.1) are also predefined.

Activity SBBs wait for events from their predecessors. The sequence in which the SBBs are activated was predefined from the code generator. The code generator extracts the order of the SBBs from the order of the BPEL activities within the BPEL process.

Figure 7 illustrates an example service with the self-controlled SBB architecture. This service consists of three SBBs, a Receive SBB, an Assign SBB, and an Invoke SBB. Two resource adaptors are used. The three SBBs are the result from the translation of a BPEL process into JSLEE. In this case the SBBs are generated from a Receive Activity, an Assign Activity and an Invoke Activity.

The service is activated when the Receive SBB receives an event from a resource adaptor. The SBB does its protocol specific communication with the resource adaptor and fires a new event to the next SBB in the end, in this case, to the assign SBB. After the assign SBB does its work (e.g. copy and set parameters), this SBB also fires an event to the next SBB. This last SBB is the invoke SBB. The invoke SBB fires events to resource adaptors and also does some protocol specific work.

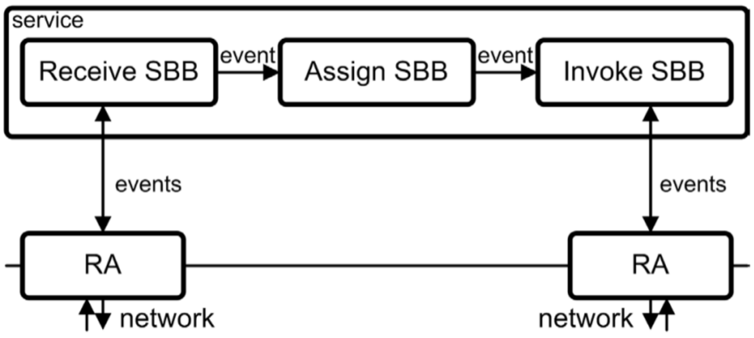


Figure 7: Service with self-controlled SBB architecture

Like many other formal languages also BPEL uses some control structures like if, while or flow. These control structures are also translated into SBBs. The if-SBB checks the if-condition and decides to which SBB the new event is forwarded. Like the if-SBB, the while-SBB checks the while condition and sends the event to the respective SBB. But in the while case, the last activity within the while returns an event back to the while-SBB. Now, the while-condition has to be checked again. This loop continues until the while-condition will turn false and the event is sent to the first SBB after the while. In case of the if- and the while-SBB the resulting event is only sent to one SBB.

The flow-SBB is able to send events to more than one SBB. Events are forwarded to all branches of the flow. The first SBB of every flow branch receives an event from the flow-SBB. With this possibility, the service gains the ability of parallel execution. This technique is described in chapter 3.2. Examples for the control structures IF, While and Flow are shown in figure 8.

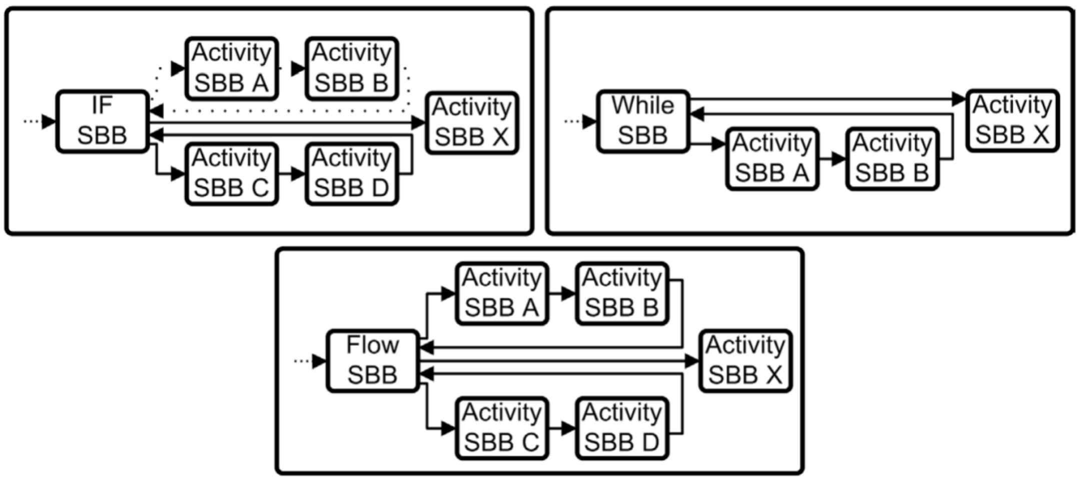


Figure 8: Control structures

5 Conclusion

Both introduced concepts avoid the problems of the original TeamCom approach. Now the services are fitting better into the JSLEE framework. They use the feature provided by the framework and offer a better expandability and modularity. This new concept copes with the problems from the old one and solved the translation of the control structures from BPEL to JSLEE. The concept with the control SBB is easier to derive from the present concept.

The state machine is used by a central SBB called control SBB. This control SBB controls all other SBBs. The concept with the most differences from the present concept is the self-controlled SBB. The exact task of a SBB is already defined during the compilation of the SBB. With this concept no communication with a central control SBB is required. The amount of events can be reduced by up to 50 percent.

This approach was derived from the original TeamCom project. It benefits from the first positive experiences with the prototypical implementation of the flow control structure (chapter 3.2). The flow implementation was the first attempt to generate parallel running activities from a BPEL process.

Based on the positive experience that was gained from the prototypical implementation of the flow, this paper extends the idea to all BPEL activities. At least one SBB is created for each BPEL activity and BPEL control structures can also be represented by an SBB.

References

[EFTG08] Eichelmann, T.; Fuhrmann, W.; Trick, U.; Ghita, B.: Creation of value added services in NGN with BPEL. In (Bleimann, U.; Dowland, P.S.; Furnell, S.M.; Grout, V.M.) *Proceedings of the Fourth Collaborative Research Symposium on Security, E-learning, Internet and Networking (SEIN 2008)*, Wrexham, UK 2008. Centre for Security, Communications and Network Research, University of Plymouth, Plymouth, UK, 2008. ISBN: 978-1-84102-196-6, pp186–193

[EFTG09] Eichelmann, T.; Fuhrmann, W.; Trick, U.; Ghita, B.: Support of parallel BPEL activities for the TeamCom Service Creation Platform for Next Generation Networks. In (Bleimann, U.; Dowland, P.S.; Furnell, S.M.; Grout, V.M.) *Proceedings of the Fifth Collaborative Research Symposium on Security, E-learning, Internet and Networking (SEIN 2009)*, Darmstadt 2009. Centre for Security, Communications and Network Research, University of Plymouth, Plymouth, UK, 2009. ISBN: 978-1-84102-236-9, pp69–80

[LET+09] Lehmann, A.; Eichelmann, T.; Trick, U.; Lasch, R.; Tönjes, R.: TeamCom: A Service Creation Platform for Next Generation Networks. In (Perry, M.; Sasaki, H.; Ehmann, M.; Bellot, G.O.; Dini, O.) *The Fourth International Conference on Internet and Web Applications and Services (ICIW 2009)*, Venice 2009. IEEE Computer Society, Los Alamitos, CA, USA, 2009, ISBN 978-0-7695-3613-2

[LRT09a] Lasch, R.; Ricks, B.; Tönjes, R.: Service Creation Environment for Business-to-business Services, *4th International IEEE Workshop on Service Oriented Architectures in Converging Networked Environments*, Bradford, UK, May 2009.

[LRT09b] Lasch, R.; Ricks, B.; Tönjes, R.: Konzept eines BPEL zu JSLEE Compilers auf Basis wieder-verwendbarer Kommunikationsbausteine. In (Tönjes, R.; Roer, P.) *Mobilkommunikation – Technologien und Anwendungen – Vorträge der 14. ITG-Fachtagung vom 13. bis 14. Mai 2009 in Osnabrück*, Osnabrück 2009. VDE, Berlin, 2009. ISBN 978-3-8007-3164-0

[OA04] OASIS Standard: *Web Services Business Process Execution Language Version 2.0*, OASIS, 2004.

[Su08] Sun Microsystems, *Open Cloud, JSR-000240 Specification*, Final Release, JAIN SLEE (JSLEE) 1.1, Sun, 2008.

[Te10] TeamCom Project Web Site (2010): http://www.ecs.fh-osnabrueck.de/teamcom.html.

[TS06] TS 23.228: *IP Multimedia Subsystem (IMS); Stage 2 (Release 5)*. 3GPP, 2006.

[W3C07] W3C: *Web Service Description Language (WSDL) Version 2.0 Part 1: Core Language*, W3C Recommendation, 2007.

[W3C08] W3C: *Extensible Markup Language (XML) 1.0 (Fifth Edition)*, W3C Recommendation, 2008.

Acknowledgment

The research project providing the basis for this publication was partially funded by the Federal Ministry of Education and Research (BMBF) of the Federal Republic of Germany under grant number 1704B07. The authors of this publication are in charge of its content.

Published in *Proceedings of the Fourth International Conference on Internet Technologies & Applications 2011 (ITA 2011)*, pp. 169-176, North East Wales Institute, Wrexham, UK, 2011, ISBN: 978-0-946881-68-0

DISCUSSION ON A FRAMEWORK AND ITS SERVICE STRUCTURES FOR GENERATING JSLEE BASED VALUE-ADDED SERVICES

Thomas Eichelmann[1, 2], Woldemar Fuhrmann[3], Ulrich Trick[1], Bogdan Ghita[2]

[1]Research Group for Telecommunication Networks, University of Applied Sciences Frankfurt/M., Frankfurt/M., Germany
[2]Centre for Security, Communications and Network Research, University of Plymouth, Plymouth, United Kingdom
[3]University of Applied Sciences Darmstadt, Darmstadt, Germany
E-mail: eichelmann@e-technik.org

ABSTRACT

This paper describes the general structure for a framework to generate value-added services. These presented service structures provide the basis for service composition within the JSLEE framework. It offers the possibility that value-added services are created, orchestrated, changed and managed from predefined and pre-deployed service components at runtime, within the JSLEE framework. The paper focuses on different approaches for the structure of service components. Several important characteristics of the structures are compared and discussed.

KEYWORDS

JSLEE, value-added services, service composition in telecommunication, service structure comparison

1. INTRODUCTION

The development of value-added services is very time consuming and experts are required to develop these services. The dream is to develop value-added services as easy as web services in the IT sector. To reach this goal, service description languages and graphical development tools may help. For the IT sector these tools already exist. A common tool to develop web services is e.g. BPEL (Business Process Execution Language) [1]. So, some projects [2] build their own tools for the development of value-added services, others try to use existing tools to generate these services [3].

This document describes the principles for an architecture to generate value-added services and focuses on the description and the analyses of considered component structures. A service can consist of one or more service components. Such a service component implements the service workflow or a part of it. Fine grained service components can be composed to complex services. In this paper JSLEE (Java Service Logic Execution Environment) [4] is used as execution environment for the services. But the discussed characteristics described in section 2.3 may also be relevant for other execution environments like Servlets or JEE (Java Platform, Enterprise Edition) Beans. The paper is structured as follows. In section 2.1 the principle structure for an architecture is defined. Four possible component structures are presented in section 2.2. In section 2.2.1 a structure is introduced which consist of a single component. From this first structure another one is derived, which supports multiple service components for parallel execution. In section 2.2.3 the orchestration of service components and in section 2.2.4 the choreography of service components are described. In section 2.3 selected characteristics of the described component structures are discussed.

2. Framework Architecture, underlying service structure and selected Characteristics

In this chapter the possible architecture for a service creation framework is described and the general elements of the architecture are presented. JSLEE is assumed as execution environment for the services. Four different service structures are presented and selected characteristics are discussed.

2.1. The framework architecture

The architecture that is presented here consists of four main elements, service creation environment, service deployment, service execution environment (SEE) and service transport layer. It is designed for service creation, deployment and execution. This is a generic architecture based on an architecture that is used in the BMBF project TeamCom [3]. In Figure 1 an overview of the generic architecture is illustrated.
The service creation environment consists of the elements, developer GUI and code generator. With the developer GUI the service developer can describe the workflow of the service. This service description has to be translated by a code generator into the language of the service execution environment. In this case Java is used as this language. The service description is translated into the Java language and a deployable unit (DU) is build from the service description. The service can be deployed on one or more application servers. It is executed in the service execution environment. The service components are managed in the component model. In section 2.2 four approaches of the component structure are described.

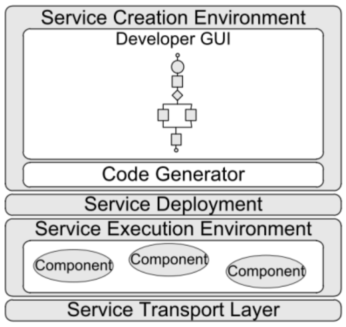


Figure 1. Framework architecture

The intention of the Graphical User Interface (GUI) is to offer the development of value-added services in a simple and fast way. Therefore the framework requires a description language that is simple but powerful enough to describe telecommunication services. Different approaches are known for a usage of a graphical user interface for the description of telecommunication services. Some approaches try to implement their own GUI e.g. [2]; others e.g. [3] use existing graphical development tools. The BPEL process in Figure 2 is an example, how value-added services can be designed with an existing tool. In this case the Oracle JDeveloper is used to build an echo service which sends incoming Instant Messages (IM) back to the sender (bpelecho_client). The process consists of three activities, a receive activity (receiveIM) which waits for incoming messages, an assign activity (Assign_Message_and_MessageReceiver) which copies the text from the received message into the new message and configures the sender and the receiver of this new message, and an invoke activity (sendbackIM) which returns the message back.

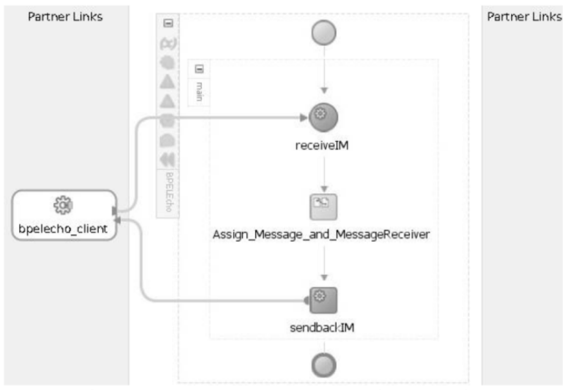


Figure 2. BPEL Process Echo Service

If the output of the GUI is not an executable value-added service, a code generator is required to convert the output of the GUI into the service code. A service description may consist of some definition files, description files and other resources which serve as input for the code generator (see Figure 3). In the case JSLEE is used as execution environment, a deployable unit is generated as output of the code generation process. The service deployment is used to copy the service to the application server. It uses the deployable unit that was generated by the code generator. The deployable unit consists of the service code, the required resources and the deployment and service descriptors. The descriptors define the service components and other parameters of the service (see Figure 3).

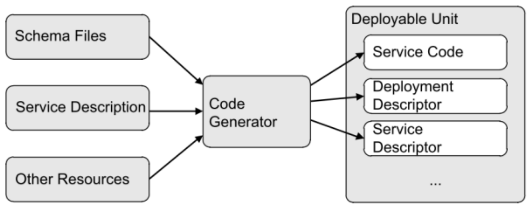


Figure 3. The Code Generator

As service execution environment the JSLEE framework is used. JSLEE is designed for ensuring low latency and providing high throughput to accomplish the requirements for communication services. The components of the service are executed in the component container. In JSLEE, these service components are called Service Building Blocks (SBBs). Four different structures of these service components are described in section 2.2.

The Service Transport Layer in the JSLEE framework abstracts different protocols in order to enable upper service layers to be independent of a specific communication protocol. Therefore it supports several communication networks. In [4] so-called Resource Adaptors (RAs) are defined abstracting the underlying infrastructure. These Resource Adaptors provide a common Java API that hides the communication protocol underneath. The communication with the SBBs is accomplished by the use of events.

2.2. The component structure

After once being generated, a service can consist of one or multiple service components. In the latter case, these components have to communicate together by exchanging events. The service components have to cooperate to ensure the fulfilment of the service workflow. In this section four different service component structures are presented, the single component structure in section 2.2.1, the parallel component structure in section 2.2.2, a structure that orchestrates service components in section 2.2.3 and a structure that uses choreography for the components in section 2.2.4.

2.2.1. Single component structure

In this concept the code generator creates only one component which represents the whole service. Within this component a state machine controls the service workflow. It decides about the events that are allowed to be received in the individual states. Figure 4 shows a scenario which represents a service that consists of only one SBB and two resource adaptors. The SBB in this figure sends and receives events from the resource adaptors. The RAs translates incoming protocol messages from the network to events and vice versa. From all three activities in the echo service in Figure 2 only one SBB will be generated.

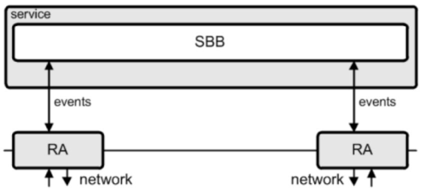


Figure 4. Single component structure example

This concept was introduced in [5], implemented and proved in [3, 6]. In principle it is possible to generate value-added services that consist of only one component, but with this approach it is not possible to realise parallel execution of service workflow elements. This characteristic leads to the new approach, the parallel component concept.

2.2.2. The parallel component approach

As described in the previous section, the problem with the execution of parallel workflow elements leads to a parallel component approach. JSLEE only supports sequential program execution in one SBB. It is not allowed to use multithreading within an SBB. A possibility to use parallel program execution in JSLEE is to use more than one SBB. They can be executed in parallel from each other. The concept to use this characteristic to obtain parallel executed service parts is described in [7]. If a parallel activity is required in the workflow, a SBB is generated for each parallel workflow part. One SBB is generated for each parallel element sequence within the workflow. The elements which are represented by one SBB are executed sequentially. The generated SBBs (see Figure 5) are communicating with the help of events.

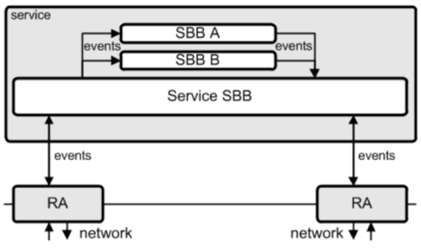


Figure 5. Parallel component structure example

If the workflow of the Service SBB reaches the position in the workflow where parallel execution is required, events are fired to the SBBs which represent the other parallel parts of the workflow. After the Service SBB has fired its events, it waits for the answer events from the called SBBs. The Service SBB uses request events to signal the other SBBs to start processing. After the Service SBB has fired its events to all parallel SBBs, it waits for the returning response events. With the request events the parallel SBBs receives the required parameter values from the main SBB. These values are used to initialise and to activate the SBBs. After processing of the SBB the changed parameter values are assigned to the response event and delivered back to the main SBB. If the Service SBB has received the response events from all parallel SBBs, the Service SBB can copy the parameter values from the events and continues processing.

2.2.3. Orchestration of service components

In the next two sections the idea of parallel service components is further extended. This section describes a component structure that allows the orchestration of service components. This concept was introduced in [8, 10], a prototype is already in development, and an enhanced framework will be published soon. In orchestration, a central process takes control over the involved services and coordinates the execution of different operations on the services involved in the operation (according to [9]). A special control SBB is required to control the service workflow and to coordinate the SBBs of the service. The control SBB assigns the work to the other SBBs, sets the required parameters, and decides, on which events an SBB has to listen and what events he has to fire. For each element of the workflow a new SBB is required. These SBBs implement the work task of the respective workflow element. For the echo service in Figure 2, three SBBs are required, i.e., one for each activity. The control SBB communicates with these SBBs via events. The control SBB is triggered by a service start event and initiates the required SBBs. The control SBB decides which SBB should be called next. In Figure 6 the control SBB is activated by the service start event. The control SBB determines which SBB is the next and fires an event with the required parameter to this SBB. The called SBB (in this case SBB A) performs its work, e.g. communicating with a resource adaptor. Subsequently the result is returned to the control SBB. With this information, the control SBB calls the next SBB according to the state machine. This procedure is repeated until the workflow is completed.

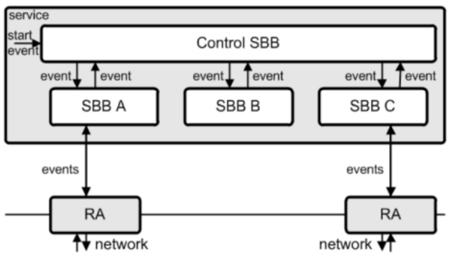


Figure 6. Orchestration component structure

This concept requires most of the intelligence in the control SBB. In the next concept, choreography is used for the service component structure.

2.2.4. Choreography of service components

Choreography does not require a central component. Each service involved in the choreography knows when to execute its operations and also knows its interaction partners (according to [9]). In this choreography based concept (introduced in [8]), the generated SBBs control themselves. No special control SBB is needed. These self-controlled SBBs communicate directly with each other. They get activated when they receive an event from its predecessor. The event includes all required parameters. So the SBB can start working after receiving the required event. Figure 7 illustrates an example service with the self-controlled SBB architecture. This service consists of three SBBs, SBB A, SBB B, and SBB C. Two resource adaptors are used. The service is activated when SBB A receives an event from a resource adaptor. The SBB A communicates with the resource adaptor and fires a new event to SBB B. After SBB B finished its work (e.g. copy and set parameters), this SBB fires an event to the SBB C. When SBB C has finished its part of the workflow, it fires an event to the resource adaptor.

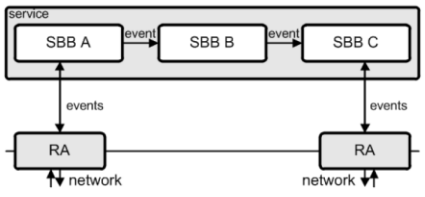


Figure 7. Choreography component structure

2.3. Selected characteristics for the component structure

All of the described concepts have their advantages and disadvantages. In this section several common characteristics of the described concepts are discussed. In Table 1 the different concepts of the component structure are compared based on selected characteristics.

Table 1. Selected characteristics of the component structure

Selected characteristics	Single Component Concept	Parallel Component Concept	Orchestration Component Concept	Choreography Component Concept
Parallel execution	No	Yes	Yes	Yes
Loose coupling	No	No	Yes	Yes
Code Generator required, precompiled	Yes	Yes	Depends on implementation	Depends on implementation
Easy expandable	No	No	Yes	Yes
3rd-Party development	No	No	Yes	Yes
Distributable service parts	No	No	Yes	Yes
Live reconfigurable	No	No	Yes	Yes

2.3.1. Parallel execution

As already mentioned in the section 2.2.1 the execution of parallel workflow elements is required in many value-added services. For a workflow which was generated with the single SBB concept only one component is created, which implements the whole workflow. In this case parallel execution of parallel parts of the workflow is not possible. The parallel component concept was specially developed to support parallel workflow execution. Also the component orchestration and the component choreography concept support parallel workflows.

2.3.2. Loose Coupling

Loose coupling means that components which are part of a service are mostly independent from each other. The first concept only consists of one component, so loose coupling is not supported. If there are no parallel sections within the workflow the parallel component approach behaves similar as the first concept and consists only of one component. Every parallel section which is added to the workflow, also adds a new component to the service. But between these service components a relatively high amount of dependencies exists. In both of the other two concepts, the orchestration and the choreography, a service consists of multiple components. Here loose coupling can be realised.

2.3.3. Code Generator required

For the single component concept and for the parallel component concept a code generator is required. The service components have to be compiled and the deployable unit has to be built with all components, the deployment descriptors, and other required resources. The orchestration concept and the choreography concept can also be compiled and put into a deployable unit. But it is also possible that the workflow will be deployed directly to the AS. There the workflow will be analysed, the required SBBs are already deployed so the service is composed from the already deployed components. The composed service can then be executed.

2.3.4. Easily expandable

The support of new protocols depends on the underlying execution environment and on the implementation of the component structure. An execution environment like JSLEE supports the new protocols by adding a new resource adaptor for the respective protocol. In the single component approach and the parallel component approach the support of new protocols is possible but these approaches require a code generator. For each protocol which has to be supported, the code generator needs to be changed. The new protocol support has, in a worst case scenario, to be implemented directly within the code generator source code. So the extension of the code generator, for a support of new protocols, may not be easy. The other two approaches do not require a code generator. In these cases the service consists of different components. Special service components implement the communication with the resource adaptors. To support a new resource adaptor and a new protocol, only these special service components have to be implemented.

2.3.5. 3rd party development

The single component concept and the parallel component concept do not really support 3rd party development. Changes have to be made in the Code Generator. The orchestration and the choreography approach both use defined service components for the communication with resource adaptors, these service components also use a defined set of interfaces which has to be implemented. These service components can be developed by 3rd party developer.

2.3.6. Distributable service parts

In the single component concept, the service consists only of one component. So it is not possible to distribute the service to other computers. If there are parallel program parts in a workflow for a service that is generated from parallel components, the distribution of service components is possible but should already be known at compilation time. The orchestration and choreography concepts consist of multiple service components which can be composed at runtime. The communication between the service components takes place via events. If some components of the service are running on another computer, this event has to be sent over the network between the components. Instead of sending an event to a local service component, the event is send to a resource adaptor. The resource adaptor translates the event into protocol data units (PDUs) and sends these PDUs to the according JSLEE application server. There, another resource adaptor receives the PDUs and translates them into an event. This event is sent to the service components.

2.3.7. Live reconfigurable

Live reconfigurable in this case means the possibility to modify the service parameters or individual service components of the services and to add or remove service components during runtime. This feature is not possible with the single component and the parallel component approach. There the service has to be compiled with all required elements and they are not changeable once the service has been deployed. The orchestration and choreography concepts allow the reconfiguration of the running service. Parameters can be modified and service components can be added or removed while the components are deployed.

3. CONCLUSION

With all of the presented approaches it is possible to develop value-added services. Also with the single component concept, services can be created, but without support of parallel execution. For the single and the parallel component concepts prototypical implementations already exists and for the orchestration and choreography concepts a prototype is in development [10]. With the new prototype other criteria such as performance issues can be evaluated. The choreography concept and the orchestration concept offer the most advantages. With a combination of the last two approaches, complex value-added services can be generated from fine grained service components at runtime. Also it is possible to dynamically reconfigure and expand the service by a rearrangement or the reconfiguration of the service components. With the presented approaches the concepts of service composition can be made available within JSLEE [10]. This allows an enhanced service creation and management framework on top of JSLEE. A prototype for this advanced framework is in development. This prototype will offer all the advantages from the choreography and the orchestration concepts and in addition, an advanced service management system, and a marketplace which offers service component sets, resource adaptors, and also value-added services. To support the reconfiguration and reorganisation of services at runtime, a web-based live GUI will also be part of the prototype.

ACKNOWLEDGEMENTS

The research project providing the basis for this publication was partially funded by the Federal Ministry of Education and Research (BMBF) of the Federal Republic of Germany under grant number 1724B09. The authors of this publication are in charge of its content.

REFERENCES

[1] D. Jordan *at all*., “Web Services Business Process Execution Language Version 2.0,” OASIS, 2007.

[2] SPICE Project Web Site (2011, February): http://www.ist-spice.org.

[3] TeamCom Project Web Site (2011, February): http://www.ecs.fh-osnabrueck.de/teamcom.html.

[4] D. Ferry, “JAIN SLEE (JSLEE) 1.1 Specification, Final Release,” OpenCloud, Sun, 2008.

[5] T. Eichelmann *at all*., “Creation of value added services in NGN with BPEL,” In *Proc. SEIN 2008*, 2008, pp186–193.

[6] A. Lehmann *at all*., “TeamCom: A Service Creation Platform for Next Generation Networks,” In *Proc. ICIW 2009*, 2009, on ICIW CD.

[7] T. Eichelmann *at all*., “Support of parallel BPEL activities for the TeamCom Service Creation Platform for Next Generation Networks,” In *Proc. SEIN 2009*, 2009, pp69–80.

[8] T. Eichelmann *at all*., “Enhanced Concept of the TeamCom SCE for Automated Generated Services Based on JSLEE,” In *Proc. INC 2010*, 2010, pp75-84.

[9] M. B. Juric, B. Mathew, P. Sarang, “Business Process Execution Language for Web Services, Second Edition,” Packt Publishing, 2006.

[10] M. Steinheimer, “Entwicklung und Bewertung von Architekturansätzen für die Kombination von feingranularen Servicekomponenten zu Mehrwertdiensten,” Master Thesis, University of Applied Sciences Darmstadt, 2011.

Published in *Proceedings of the 3rd International Conference on Internet and Applications 2012 (ITAP 2012)*, Wuhan, China, 2012, ISBN: 978-1-4577-1575-4

A JSLEE based Service Creation and Service Delivery Framework for value-added services in Next Generation Networks

Thomas Eichelmann, Ulrich Trick
Research Group for Telecommunication Networks
University of Applied Sciences Frankfurt/M
Frankfurt/M, Germany
{eichelmann, trick}@e-technik.org

Woldemar Fuhrmann
Department of Computer Science
University of Applied Sciences Darmstadt
Darmstadt, Germany
woldemar.fuhrmann@h-da.de

Bogdan Ghita
Centre for Security, Communications and Network Research
Plymouth University
Plymouth, United Kingdom
bogdan.ghita@plymouth.ac.uk

Abstract— **This paper describes a framework for an easy and timesaving generation of value-added services in Next Generation Networks and introduces new possibilities for the service creation and service execution through an extension of the common telecommunication real-time execution environment JAIN SLEE. Value-added services are described with graphical development tools. This offers an easy and fast graphical service design. Through the introduction of extensions to the execution environment the service is automatically composed from predefined and pre-deployed components. These extensions also introduce new possibilities for the composition, execution, management, and reconfiguration of value-added services.**

JSLEE; value-added services; service creation; BPEL; NGN

I. Introduction

The telecommunication market always demands new value-added services. New service ideas have to be developed and offered to the market before their competitors can react. This requires that the development cycles of the value-added services are as short as possible. Complex communication protocols and complex service development leads to the fact that experts are required for the development of value-added services. But to be able to respond to the needs of specific customer groups, the development of customer specific services should be possible also for non expert developer. Furthermore, a Service Delivery Platform (SDP) must satisfy particular requirements, which exclude the usage of web-services or similar approaches. An SDP which offer multi-protocol support together with a high-throughput and low-latency environment is required.

The solution presented in this paper introduces a service creation and service delivery environment that solves the addressed problems. The Service Creation Environment (SCE) offers graphical user interfaces to describe the value-added service and hide the underlying heterogeneous communication networks. Therefore, the developer does not need any detailed knowledge of communication protocols and is able to focus on the application logic instead. For the service description a language that has been optimized for business processes is suggested: the Business Process Execution Language (BPEL) [1]. BPEL however has not been developed for real time communication services in heterogeneous networks. Therefore, from the business process description a value-added service is generated. The SCE allows a reconfiguration of the service logic even during the execution of the service. The SDP introduces extensions to the Java APIs for Integrated Networks (JAIN) Service Logic Execution Environment SLEE [2]. JAIN SLEE also called JSLEE is a common telecommunication real-time execution environment. The extensions offer an automated service generation, extended service management capabilities, a service execution layer, and a layer which handles the resources and the protocols. This leads to new opportunities for rapid and efficient service creation using a new SCE with higher level of abstraction and automated service generation.

The paper is structured as follows. Chapter II starts with a discussion on some related work in this area of research. Chapter III offers a short overview of JSLEE and discusses the advantages and disadvantages of JSLEE. Chapter IV introduces the new Service Creation and Service Delivery Framework with the new layered structure within the service creation and service delivery framework and gives an example how services are created and executed. The paper ends with the conclusion in chapter V.

II. Related Work

In the work: “Orchestration in Web Services and Real-Time Communications”, L. Lin and P. Lin [3] describes an approach to enable a service creation environment for complex (orchestrated) real-time communication services through a service broker on top of Next Generation Networks (NGN).

The goal of the research is to combine the emerging Web/telecommunications service space. They take a hybrid approach for converged voice-data application by adding Web service interfaces to real-time communication flows. This Web services can be orchestrated along with other Web services in BPEL. This work shows an approach how communication services can be composed with BPEL. In our approach BPEL is only used for the service description of the service. In contrast, no real BPEL engine is required. We also offer a framework for an automated generation, execution, configuration and management of the service.

In the paper: “Creating Value Added Services in Internet Telephony: An Overview and a Case Study on a High-Level Service Creation Environment” [4] a case study of a high level graphical SCE is described. It consists of a graphical user interface (GUI). A service can be developed from the eight functions: Start, Timer, Call, Loop, Join, Sync, Play and End. The service logic can be developed with these functions within the GUI. The scope of the experiment is limited to services that originate calls, like Wake-up Call, Call Center or Third Party Call. The services can be executed in a self-developed SLEE that is based on a Parley/Session Initiation Protocol (SIP) Gateway implementation. Therefore the experiment only supports the SIP protocol and only allows basic telephony services. Furthermore the services have to be compiled before execution and cannot be composed from existing components.

III. Discussion on the JSLEE Framework

To understand the proposed extension of the JSLEE framework, the foundations of JSLEE should be described and the strengths and weaknesses should be clarified. JSLEE is a high throughput, low latency event processing application environment for communication services. It allows the implementation of scalable high available communication services. The access to network resources is offered by Resource Adaptors (RAs). The service components are called Service Building Blocks (SBBs) “Fig. 1”. A service consists of one or more SBBs. These SBBs are deployed on the JSLEE Application Server. Invocations are transmitted asynchronously via events. The Service Transport Layer abstracts different protocols in order to enable upper service layers to be independent of a specific protocol. Therefore it supports several communication networks.

RAs abstract the underlying infrastructure. When a communication protocol message is received, the corresponding RA translates this message into a Java event class. Afterwards the event is passed to a JSLEE element that is called event router. The event router looks up the services that are subscribed for the specific event and delegates the event to these services. Accordingly the service itself is able to react on the event and to create an answer by using defined Java interfaces. The answer is translated to a communication protocol response by a RA.

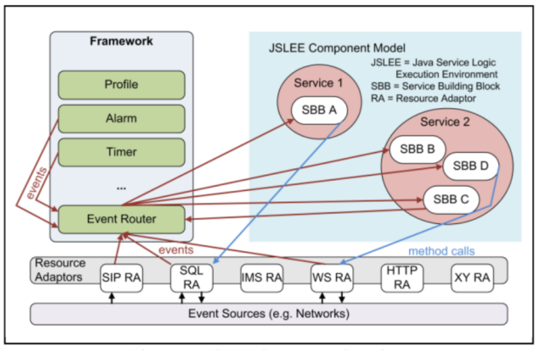


Figure 1. JSLEE Component Container

As described this framework is ideal for value-added services. But there are some problems which also need to be discussed. The development of JSLEE-based services is very complex. The developer of a value-added service requires expert knowledge of java, JSLEE and of the underlying protocols. The required experts are rare and expensive. Developers which are new to JSLEE suffer from the steep learning curve. This leads to an expensive and long development of new services.

IV. Service Creation and Service Delivery Framework

The architecture is designed to provide advanced possibilities for service creation and execution. It is divided into the layers: Service Creation Environment (SCE), Service Execution Environment (SEE) and the Service Transport Layer (STL) “Fig. 2”. Parts of this architecture are based on the architecture developed in the TeamCom project [5].

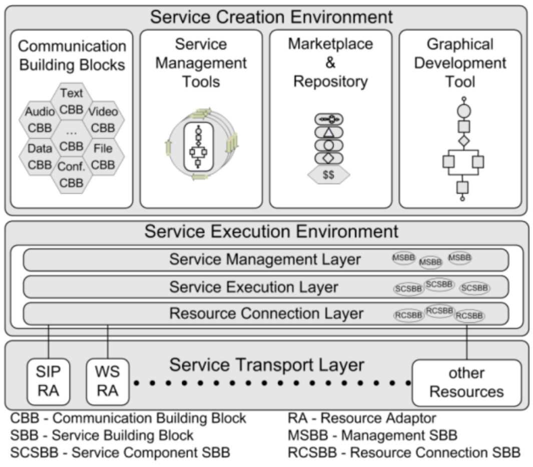

Service Creation Environment
Communication Building Blocks
Text CBB
Audio CBB
Video CBB
...
Data CBB
File CBB
Conf. CBB
Service Management Tools
Marketplace & Repository
$$
Graphical Development Tool
Service Execution Environment
Service Management Layer
MSBB
Service Execution Layer
SCSBB
Resource Connection Layer
RCSBB
Service Transport Layer
SIP RA
WS RA
other Resources
CBB - Communication Building Block
SBB - Service Building Block
SCSBB - Service Component SBB
RA - Resource Adaptor
MSBB - Management SBB
RCSBB - Resource Connection SBB

Figure 2. The Framework architecture

A. *The Service Creation Environment*

The Service Creation Environment offers tools to design the services and to acquire new services and other resources. It consists of a repository of Communication Building Blocks (CBBs), the service management tools, and a repository with a marketplace interface to acquire new services and service components. The SCE also offers graphical development tools to describe the services. BPEL is used as service description language.

BPEL is an established business process specification language normally related to web services. In this case it is used to describe value-added services. It allows the service developer to use existing graphical BPEL design tools to design the service logic. The framework parses the created BPEL file and composes the value-added service from precompiled and already deployed service components. The CBBs are the logical description of the available functionalities. In BPEL these CBBs are available as partner links. They offer a representation for the available resources and functionalities. If a service requires a special functionality, e.g. calling a participant of a conference, the corresponding method from the partner link that handles conferencing issues has to be invoked in the BPEL process.

B. *The Extended Service Execution Environment*

To solve the described problems with JSLEE (chapter III) this paper proposes an extension to the JSLEE framework which solve these problems and offer a fast and cost efficient development of value-added services. Three new layers are defined within the service component container of the JAIN SLEE Application Server (AS) which enables the support for an automated composition, configuration, and management of value-added services "Fig. 3". The three new layers are the Service Management Layer, the Service Execution Layer and the Resource Connection Layer.

The Service Management Layer consists of management tools. The service management tools control the lifecycle of the value-added services. They offer the functionality to start and stop the services. They can also generate new services from a BPEL process by composing existing components. These tools are implemented as SBBs. To distinguish them from the other SBBs they are called Management Service Building Blocks (MSBBs). The service management supports the control of the available services. The most important MSBBs are the Service Description Parser and the Service Control MSBBs (SCMSBBs) "Fig. 3". An SCMSBB is assigned for the composition, configuration, and lifecycle control of the required service components of a service.

In the Service Execution Layer the implementation of the service logic is located "Fig. 3". This logic is represented by the Service Component SBBs (SCSBBs). The SCSBBs realize the service logic which was described in the workflow of the BPEL process [6]. They are configured and controlled by a SCMSBB. The SCMSBB configure the SCSBBs by setting the required parameters, and decides on which events a SCSBB has to listen and what events it has to fire.

The Resource Connection Layer offers the interfaces to the resources. Special SBBs called Resource Connection SBBs (RCSBBs) form the connection between the service logic and the RAs. RCSBBs implement the methods which are described in the CBBs. They map the logical method description to the functionality which is offered by the RAs. In the service that is generated from the BPEL process, the RCSBBs are called by the SCSBBs to invoke the real functionalities.

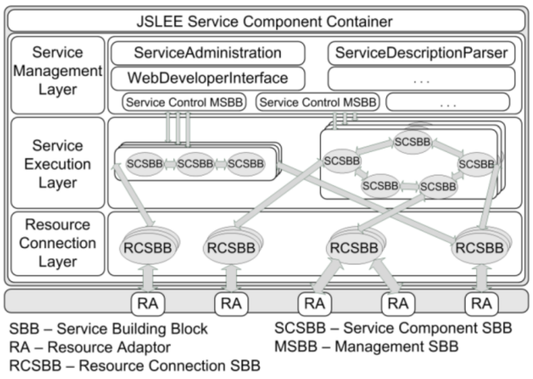


Figure 3. Service Execution Environment

C. *Service creation and execution*

As input for the service creation a service description is required. This description is analyzed by a Service Description Parser. BPEL is supported per default for the service description by the framework. To support non BPEL service descriptions, new Service Description Parsers have to be added to the Service Management Layer "Fig. 4". The SCMSBB composes the required SCSBBs and sends events to configure them. The configuration consists of the information from which component the SCSBB can receive events and what is to do with the received events. The structure of the generated service is derived from the structure of the service description. For each element from the service description a corresponding SCSBB is required "Fig. 5". To signalize that a SCSBB is created and configured, the SCSBB sends an event to the SCMSBB. If all components of the service are ready, the service is ready for execution.

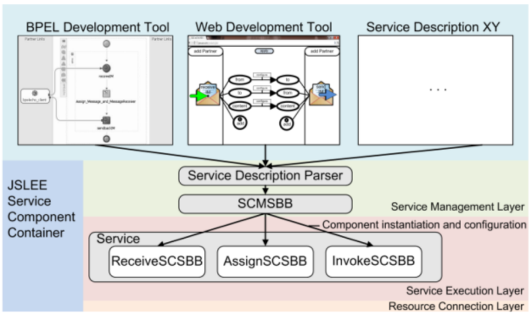


Figure 4. Service Composition

The service that is described in “Fig. 5” waits for incoming mails. If a mail is received, the content of the mail is assigned to a new mail, to an instant message, and to a SMS. Then, all messages are sent out. For each element in the service description (upper part of “Fig. 5”) a corresponding SCSBB in the composed service (lower part of “Fig. 5”) exists.

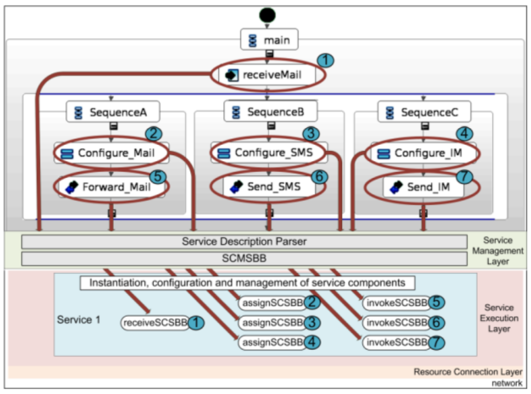


Figure 5. Service Composition Phase

In the execution phase of the service “Fig. 6” all involved components are already configured for execution. When a mail is received by the RA, it fires an event to the appropriate RCSBB and the service is triggered. If the service management decides that a service should be stopped, it fires a service stop event to the SCMSBB of the service. The SCMSBB also fires service stop events to all SCSBBs involved in the service. These SCSBBs answer with confirmation events before they destroy their instance. If all SCSBBs of the service have sent the confirmation, the SCMSBB also sends a confirmation event to the management and removes the service from the AS.

The framework also supports the reconfiguration of running services. If a service should be reconfigured, the management generates a reconfiguration event and fires this event to the SCMSBB of this service. The SCMSBB analyses the changes in the service workflow and initiates the reconfiguration of the involved service components.

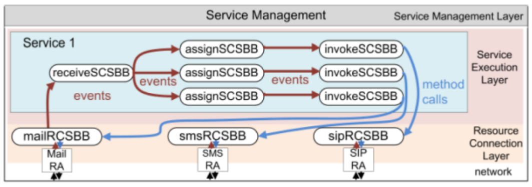


Figure 6. Service Execution Phase

V. Conclusion

This paper proposes a new service creation and service delivery framework. It is based on the JSLEE framework and offers all advantages of the JSLEE framework like a high throughput, low latency, and multi protocol architecture. Additionally the framework offers a simple and fast service development. The services are described with a graphical BPEL development tool. The JSLEE service is automatically composed from the BPEL description. The BPEL elements represent the service logic i.e., the workflow of the service. These BPEL activities have their counterparts in JSLEE as SCSBBs. The required resources/functionalities are described in BPEL as partner links. The service developer does not need special knowledge of the underlying protocols. The required functionalities have only to be invoked in BPEL. Self developed functionalities and resource adaptors can be integrated into the framework by mapping the resources in the RCSBBs to the functionalities described in the CBBs. All required components like services, RAs, CBBs, RCSBBs can also be acquired from the marketplace of the framework. This allows 3rd party developers to offer own resources and services [7]. New protocols can be supported easily by providing the corresponding RAs, RCSBBs and CBBs. The developed framework prototype already demonstrates the capabilities of the approach. The most important framework elements are implemented. A BPEL service description parser, HTTP and (basic) SIP support are also available. Value-added services which use the implemented components can already be described with BPEL and generated and executed in the framework.

Acknowledgment

The research project ComGeneration providing the basis for this publication was partially funded by the Federal Ministry of Education and Research (BMBF) of the Federal Republic of Germany under grant number 1724B09. The authors of this publication are in charge of its content.

References

[1] OASIS Standard, “Web Services Business Process Execution Language,” Version 2.0, OASIS, 2007.

[2] Java Specification Requests, “JSR 240: JAIN SLEE (JSLEE) 1.1 Specification, Final Release,” Sun Microsystems, OpenCloud, Sun, 2008.

[3] Lin, L., Lin, P. (2007), “Orchestration in Web Services and Real-Time Communications,” IEEE Communication Magazine, 07 2007.

[4] R.H. Glitho, F. Khendek, A. De Marco, “Creating Value Added Services in Internet Telephony: An Overview and a Case Study on a High-Level Service Creation Environment,” in Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions, pp 446 - 457, 2003.

[5] A. Lehmann, T. Eichelmann, U. Trick, R. Lasch and R. Tönjes, “TeamCom: A Service Creation Platform for Next Generation Networks,” in The Fourth International Conference on Internet and Web Applications and Services (ICIW 2009), M. Perry, H. Sasaki, M. Ehmann, G. O. Bellot, and O. Dini, Eds., ISBN 978-0-7695-3613-2, pp.12-17, IEEE Computer Society, Washington, DC, USA, 2009.

[6] T. Eichelmann, W. Fuhrmann, U. Trick, and B. V. Ghita, “Enhanced Concept of the TeamCom SCE for Automated Generated Services based on JSLEE,” in Proceedings of the Eighth International Network Conference (INC 2010), U. Bleimann, P. S. Dowland, S. Furnell, and O. Schneider, Eds., ISBN: 978-1-84102-259-8, pp75-84, University of Plymouth, Plymouth, 2010.

[7] T. Eichelmann, W. Fuhrmann, U. Trick, and B. V. Ghita, “Discussion on a Framework and its Service Structures for generating JSLEE based Value-Added Services,” in Proceedings of the Fourth International Conference on Internet Technologies and Applications (ITA 11), S. Cunningham, N. Houlden, V. Grout, D. Oram, and R. Picking, Eds., pp 169-177, ISBN: 978-0-946881-68-0, Glyndwr University, Wrexham, 2011.