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].

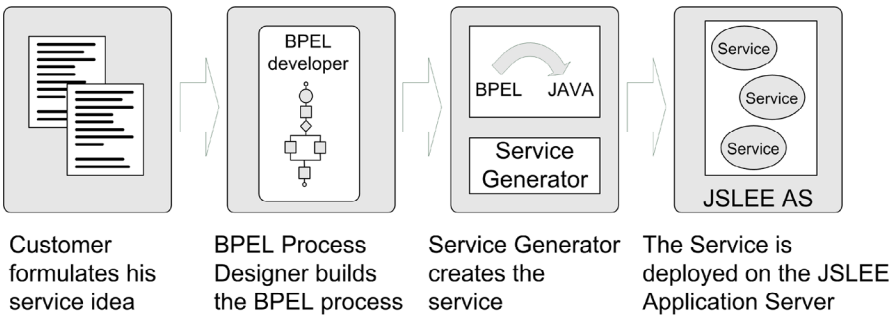


Figure 1: TeamCom service development

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.

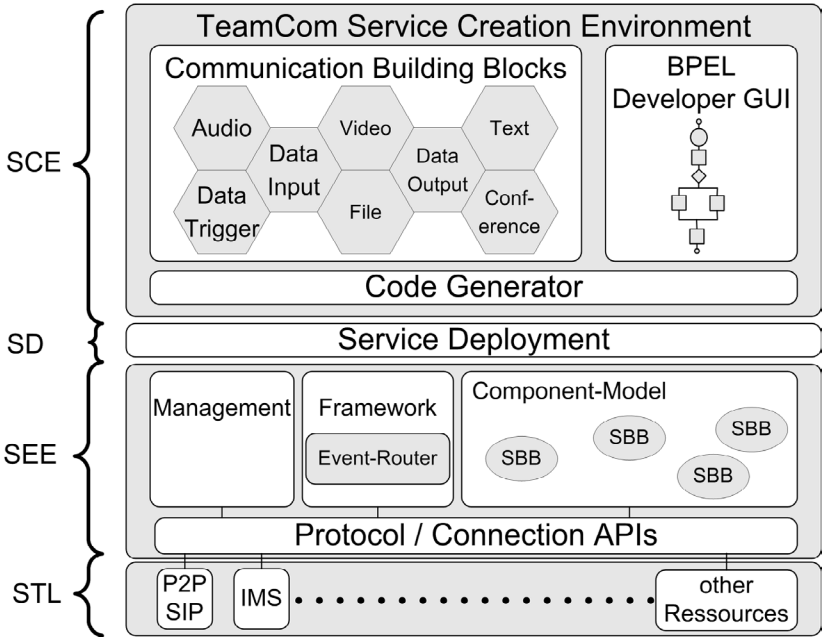


Figure 2: TeamCom Architecture

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.

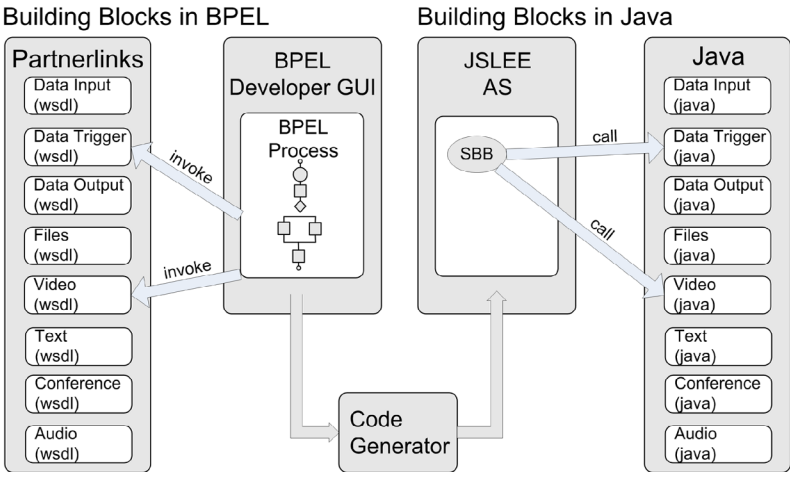


Figure 3: Representation of Communication Building Blocks

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.

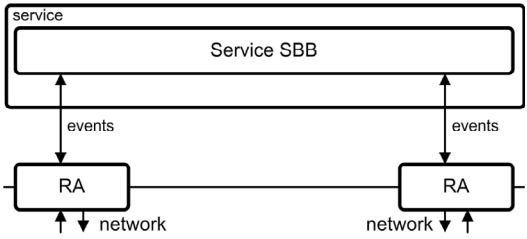


Figure 4: Monolithic TeamCom approach

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.

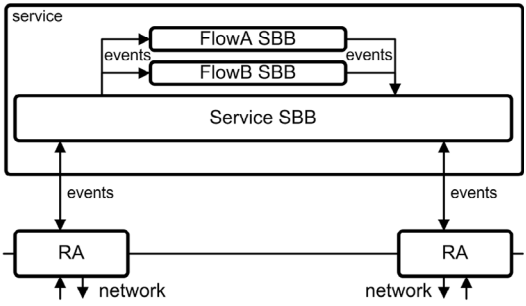


Figure 5: Monolithic TeamCom approach with flow

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.

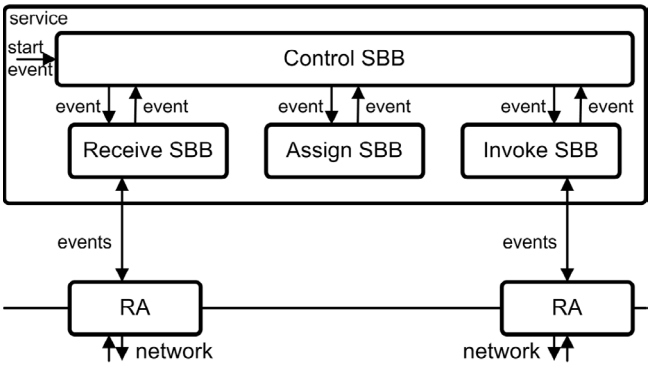


Figure 6: Service with controlled SBB architecture

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.

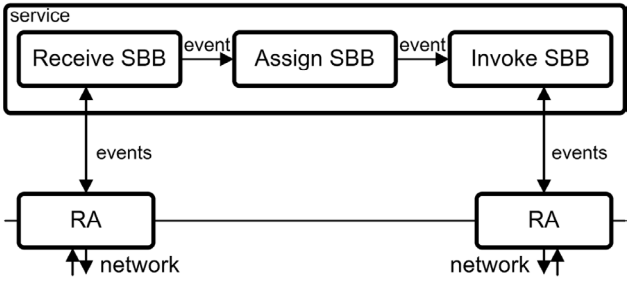


Figure 7: Service with self-controlled SBB architecture

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.

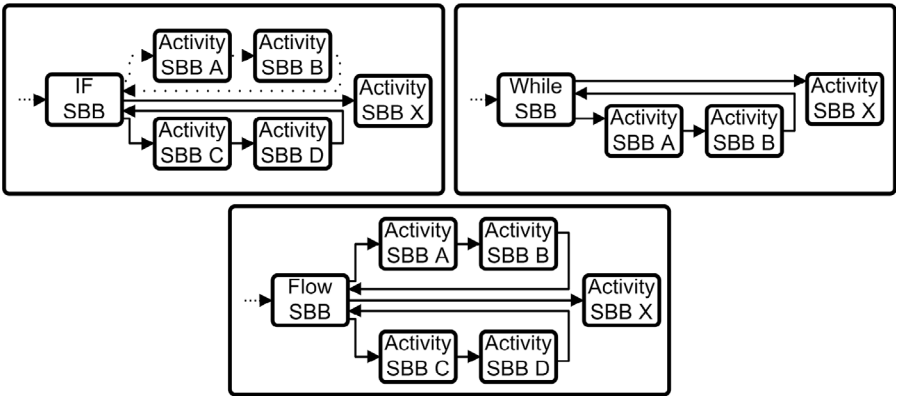


Figure 8: Control structures

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.