Institut für Architektur von Anwendungssystemen
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Studienarbeit Nr. 2406

Unified Service-Composition for BPEL

Bing Shao

Course of Study:	Computer Science
Examiner:	Prof. Dr. Frank Leymann
Supervisor:	Dipl.-Inf. Katharina Görlach
Commenced:	Novermber 08, 2012
Completed:	May 10, 2013
CR-Classification:	D.2.11, D.3.2, F.3.2, F.4.2, H.4.1

Abstract

Nowadays there are more and more requests from life and business. Those requests can be realized by composing the existed services. There are multiple models can specify services. To compose the services which are specified by different models, it's necessary to unify those models. Formal grammar is used as unified model. Görlach presents a concept to transform existing service composition models to a unified model. Based on the work, the control flow based language, BPEL is focused on in this thesis. This work implements the transformation of BPEL to formal grammar. The architecture of implementation is introduced. To test and verify the implementation test cases are represented.

Contents

List of Figures

List of Listings

1. Introduction

In recent years the web has changed the way we live. To satisfy people's requests, many services are running on the web. Man can use them without understanding how they working. For the business, it's easier to interchange information with partners. Sometimes the needed services are not existed. A definite way is to implement a new service. There is also a efficient solution. By choosing the existed services, put them in logical order and compose them as the new service. Using service composition, the developing time of new service and the cost of developing can be less. It's helpful to increase business benefit and decrease maintain cost.

As a approach to specify business process behavior, the second version of Business Process Execution Language (BPEL) was announced in 2007 [AAA+07]. As a standard language in this area BPEL has advantage, more acceptability from industry. But it's not the only way to describe business process. There are other models to depict business process, such as ConDec [PA06], Scufl [OGA+05]. It's possible, more than one language will be taken by industry in the future. That's means, those languages should understand each other, furthermore communicate, integrate. Use formal grammar the meta-model of a language can be have, not only for control flow based meta-model, such as BPEL, but also for constraint based meta-model, such as ConDec. With the help of the formal grammar, a unified model is a solution. After transformation of BPEL to formal-grammar-based unified model, the executability of the business process should be verified.

The work in this thesis focused on unified service-composition for BPEL. As figure 1.1 shows the input is BPEL file, as grammar based unified model the formal grammar of BPEL process is outputted, and the grammar should be executable by a automaton. The related web service

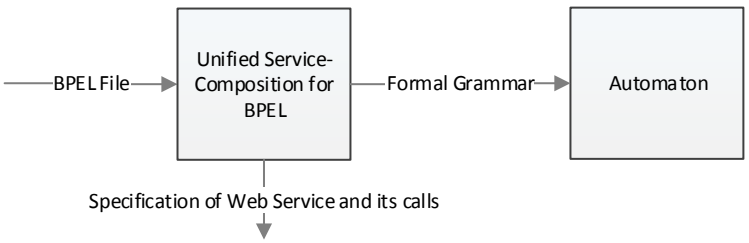


Figure 1.1.: In- and Output of Unified Service-Composition, BPEL file is input, the output includes the formal grammar and specification of web service and its calls

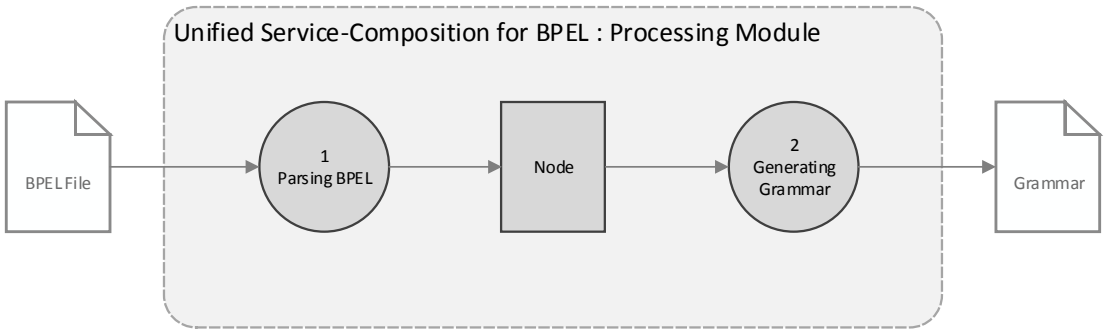


Figure 1.2.: The Overview of the Processing Module

and the calls of the service should be specified. The figure 1.2 shows the overview of the processing module. The first step is Parsing BPEL, after parsing the information of BPEL process, which described in BPEL file, are saved in a intermediate model, `Node`. The second step is Generating Grammar, this step transform the BPEL process to the formal grammar.

This work is structured as follows:

Chapter 2 – Background: The background knowledge that are necessary for this thesis is introduced.

Chapter 3 – Parsing BPEL: At the beginning read a BPEL file and parse the BPEL process, then the needed information for the further work is saved in a intermediate model, `Node`.

Chapter 4 – Implementation of Generating Grammar: The implementation of this work and the transformation for BPEL to formal grammar are represented. Test cases are used to test and verify the implementation.

Chapter 5 – Summary: The conclusion of this thesis will be described.

2. Background

In this chapter the needed background knowledge for reading this thesis are briefly introduced.

Business Process Management / Workflow Management

A Business Process Management definition is: “Supporting business processes using methods, techniques, and software to design, enact, control, and analyze operational processes involving humans, organizations, applications, documents and other sources of information.” [AHW03]. A Workflow Management System (WFMS) is defined as: “A system that defines, creates and manages the execution of workfows through the use of software, running on one or more workflow engines, which is able to interpret the process definition, interact with workflow participants and, where required, invoke the use of IT tools and applications.” [AHW03].

The Figure 2.1 shows the relationship between workflow management and business process management using the BPM life cycle. The phases in the life cycle support operational business processes. The processes are in design phase designed. In configuration phase, the designed processes are implemented by configuring the process cognizant of a system, e.g. workflow management system. The enactment phase decide where the operational processes are executed. Diagnosis phase analyze the processes to identify problems. [WAV04]

Web Services Business Process Execution Language

Web Services Business Process Execution Language (BPEL) [AAA+07] is a kind of XML-based programming language, used to describe the business process. The target of BPEL is

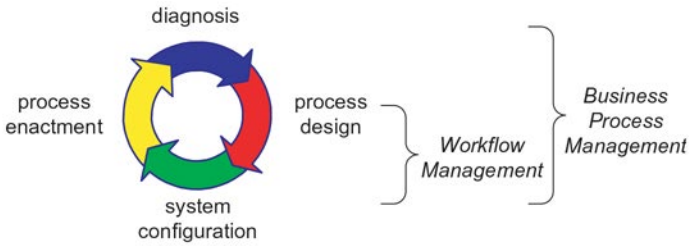


Figure 2.1.: The BPM life cycle to compare WFM and BPM [WAV04].

standardization of business process definition. A BPEL process can be composed by one or more activities. There are two kinds of activities in BPEL, Basics Activities and Structured Activities. Some steps of the described business process are implemented by web service.

BPEL is dependent on the following standard technologies:

- WSDL: is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. [CCM+01]
- XML: is a subset of SGML that is completely described in this document. Its goal is to enable generic SGML to be served, received, and processed on the Web in the way that is now possible with HTML. [BPSM+97]
- Xpath: is a language for addressing parts of an XML document, designed to be used by both XSLT and XPointer. [CD+99]
- WS-Addressing: provides transport-neutral mechanisms to address Web services and messages. [BCC+04]

WSDL [CCM+01] is the most important in those technologies, because BPEL is dependent on web services. A BPEL process can be published as a WSDL file for web service, and can be called by other web serves. Note that, BPEL does not support human-machine interaction directly. The process, which is described by BPEL, communicate only with web services, and those web services support information interchange with users. The process written by BPEL can work on the platform or product supporting the BPEL standard.

BPEL supports two kind of business processes, executable and abstract. Executable business processes define concrete tasks and the needed service calls, can be executed. Abstract processes describe information interchange between web services, but not behavior details. In this thesis the executable business process is covered.

Web Services

The W3C definition is “A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.“ [Fer04].

Like BPEL Web Service Description Language (WSDL) is also an XML-based standard. In BPEL process we use WSDL to call Web services. BPEL process can also be represented as WSDL for calling. WSDL document uses the following elements in the definition of network services: [CCM+01]

- Types: a container for data type definitions using some type system (such as XSD).
- Message: an abstract, typed definition of the data being communicated.

- Operation: an abstract description of an action supported by the service.
- Port Type: an abstract set of operations supported by one or more endpoints.
- Binding: a concrete protocol and data format specification for a particular port type.
- Port: a single endpoint defined as a combination of a binding and a network address.
- Service: a collection of related endpoints.

Based on the specification, to define a functionality of Web service, the endpoint of service is necessary. In order to get the message format and protocol details of service, binding should be known. Obviously the operation is needed, and as the connection point of the functionality port type is essential. That means, using endpoint of service, binding, port type and operation, a functionality of service is specified. To call it, the type of parameters should be defined in message.

Formal Grammar

A formal grammar is defined as the tuple (V, Σ, P, S). The meaning of four parts are:

- V : a finite set of non-terminal symbols.
- Σ : a finite set of terminal symbols. And $V \cap \Sigma = \emptyset$.
- S : Start symbol. $S \in V$.
- P : a finite set of production rules. the Form of rules is $(\Sigma \bigcup V)^* V (\Sigma \bigcup V)^* \rightarrow (\Sigma \bigcup V)^*$.

Automata

Automata is a abstract computing device that takes a word as input and decides either to accept it or reject it. A automata is represented formally by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where:

- Q is a finite set of states.
- Σ is a finite set of symbols, called the alphabet of the automata.
- δ is the transition function, that is, $\delta : Q \times \Sigma \rightarrow Q$.
- q_0 is the start state, which means, the state of the automaton before any input has been processed, where $q_0 \subseteq Q$.
- F is a set of states of Q (i.e. $F \subseteq Q$) called accept states or end states.

3. Parsing BPEL

As figure 3.1 shows, parsing BPEL is the first step of the processing module. In this step BPEL file as input is read and parsed. After parsing, the taken data from input are saved into the intermediate model, `Node`. `Node` represents all kind of activities and scopes of BPEL processes, such as the basic activities (e.g. `invoke`, `assign`), structured activities (e.g. `sequence`, `flow`), scopes (e.g. `scope`, `eventHandler`). Except them, a utility class `NodeFactory` is defined for class `Node`, in order to manipulate `Node` and its subclass easier. The second step, generating grammar is introduced in chapter 4.

In this chapter, the class `Node` and its subclass is presented in section 3.1. The section 3.2 shows, how the parsing method is chosen. At the end, the class `NodeFactory` is introduced in the section 3.3.

3.1. Class Node and Subclass

Before describe the parsing method, class `Node` should be introduced. `Node` is the fundamental data structure in this work. After parsing, all the information of a XML node in BPEL file will be saved as data members of class `Node`. And there are additional data members (cf. figure 3.2), which to be used in the further step. Among those data members, `Children` is one of the most important. It saves all the child nodes of current node. In order to save the child

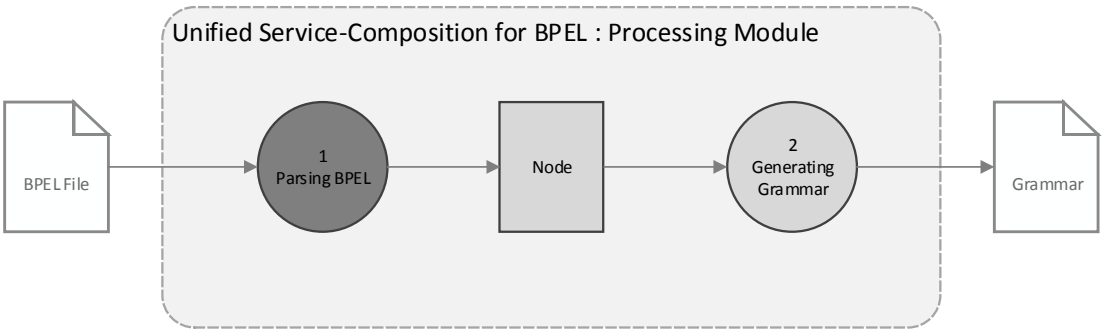


Figure 3.1.: The Overview of the Processing Module with Parsing BPEL (Step 1) emphasized

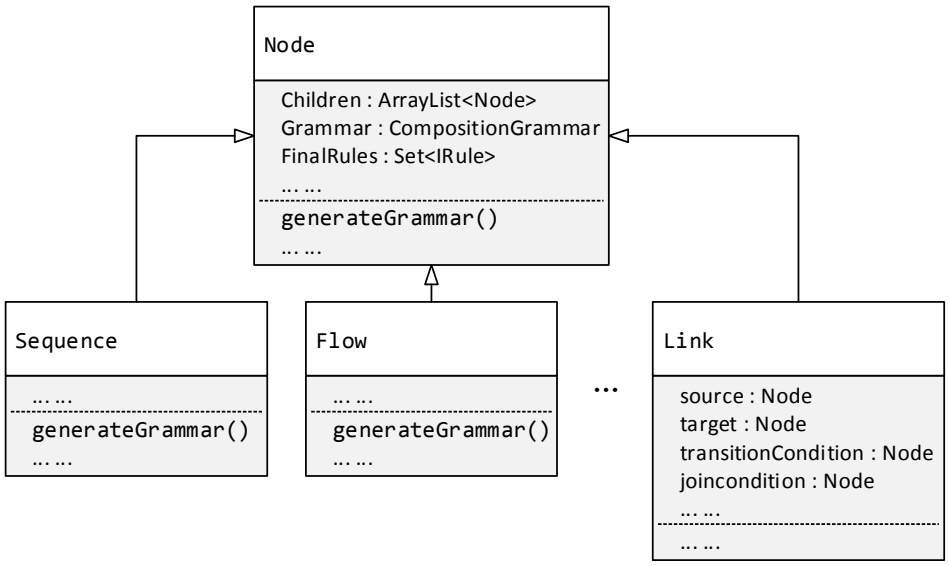


Figure 3.2.: The Overview of Class Node and Subclass

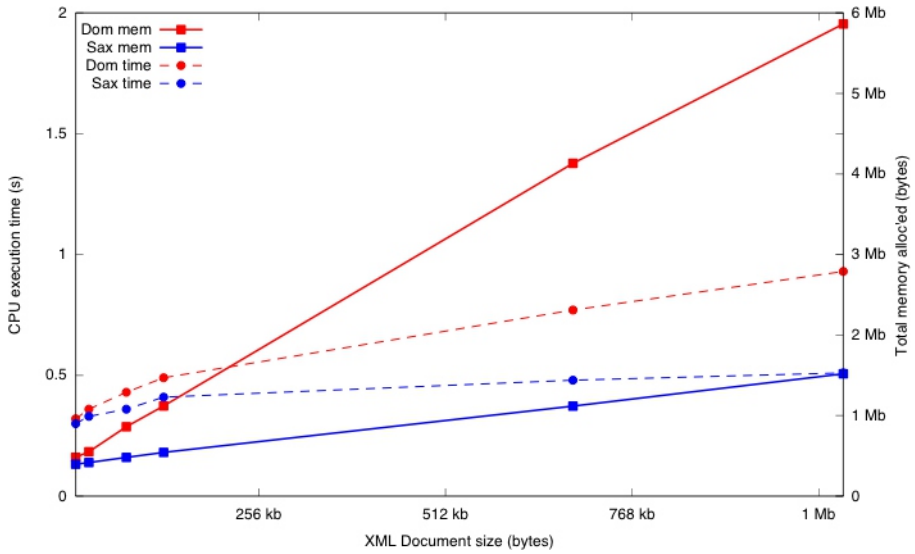
nodes in the same order as the BPEL file, `Children` is defined as type `ArrayList`. Use data members `Children`, the original BPEL document will be constructed as a tree model.

Most of method members in class `Node` are simple, except GENERATEGRAMMAR. The method is the core of this work. In this method, the grammars of all child activities should be generated first. Based on its specification, the grammars of child activities are merged. Each kind of subclass has its own principle, its grammar to generate, therefore the subclass of `Node` was defined, such as class `Sequence`, `Flow`, etc. Using GENERATEGRAMMAR generated formal grammar is saved in `Grammar` (cf. figure 3.2). Its type, `CompositionGrammar` is described in figure A.1.

In the generated grammar of a BPEL activities or scopes exists the production rules, which produce terminal state of the grammar, and the terminal state means the end state of the activities or scopes. Those production rules are saved in `FinalRules` (cf. figure 3.2). It will be used for the merging of grammar in the next chapter. For example, the `FinalRules` of the following grammar is the rule (3).

$$\begin{array}{lll} (1) & A & \longrightarrow aB \\ (2) & B & \longrightarrow bC \\ (3) & C & \longrightarrow c \end{array}$$

Link



.

Figure 3.3.: Java XML parsing performance using SAX and DOM [Gor07]

`Link` is a special subclass of class `Node`. In WS-BPEL, `<link>` as child element works with `<flow>` activity together, it express the synchronization dependencies of activities inside of the `<flow>`. Using the optional element `<source>` and `<target>`, the synchronization relationships between activities are established by `<link>`'s.

The `<source>` activity has an optional activity `<transitionCondition>`, which its value is a *boolean* expression. It looks like a trigger for a `<link>`. When the value of `<transitionCondition>` is true, it's `<link>` will be activated. Similarly, on the other side of `<link>`, the collection element of `<target>`, `<targets>` has an optional element `<joinCondition>`. The activity will be triggered, only if the value if its `<joinCondition>` is true. If no `<joinCondition>` is specified, the `<joinCondition>` is the disjunction (i.e. a logical OR operation) of the link status of all incoming links of this activity. [AAA+07]

The functionality of `<link>` is the reason why class `Link` need the additional data members (cf. figure 3.2). `source` saves the `Node`, which is specified in its optional element <source>. Like `source`, `target` is for element `<target>`, `transitionCondition` is for element `<transitionCondition>`, `joinCondition` is for element `<joinCondition>`. And `Link` does not has its own formal grammar, its functionality can be represented in the grammar of `Flow`. Which means, its method `generateGrammar` does not need to be specified.

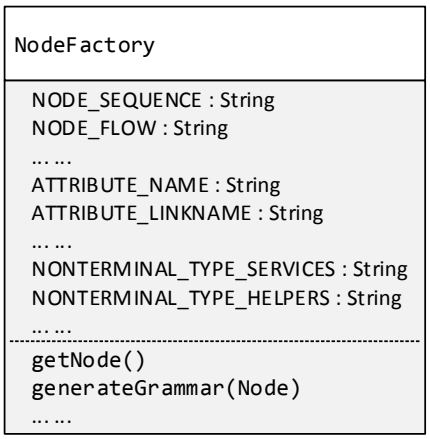


Figure 3.4.: The Overview of Class NodeFactory

3.2. Choice of Parsing Method

Usually there are two kind of XML parsers in java, DOM [C+01] and SAX [M+98].

DOM is tree model. The entire XML is parsed and a DOM tree (of the nodes in the XML) is generated and returned. Once parsed, the DOM tree can be navigated to access the data embedded in the nodes. Typically the DOM approach is used for small XML structures that may need to be modified and accessed in different direction once loaded.

SAX is event based. Events are triggered when XML is being parsed. For example, event startDocument is raised when at the beginning of XML document, and event endDocument for end of document. SAX reads XML as steam, doesn't need the whole XML to be loaded previously.

In general, DOM can manipulate entire XML, traverse in any direction, is easier to use, but has an overhead of parsing XML before starting. SAX use a tiny part of memory, runs faster, but traverse node by node, and read only. Onne Gorter has presented performance of XML parsing using SAX and DOM in java [Gor07], which is observe the difference clearly. As figure 3.3 shows, using DOM and SAX parsing small document, the overhead of memory and time is similar. But when document size increased to 1Mb, DOM takes twice the time, and 5 times the memory compared to SAX [Gor07]. In this thesis, we don't need to modified the input BPEL process, therefore SAX has it's advantages, to be chosen.

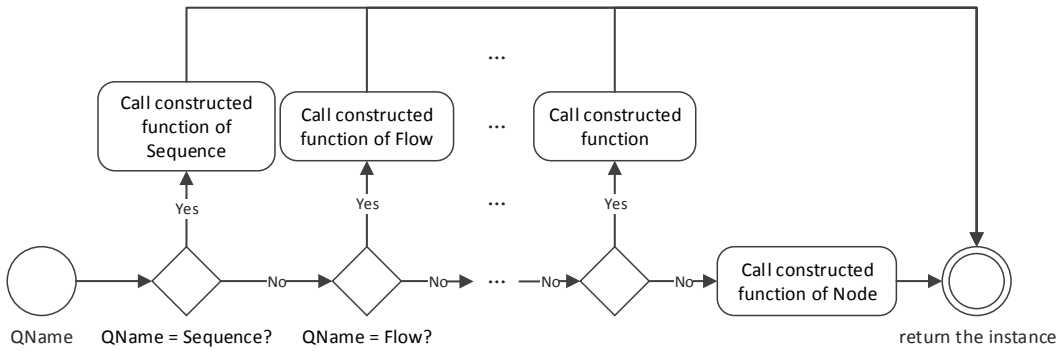


Figure 3.5.: The Activity Diagram of method getNode

3.3. Utility Class NodeFactory for Class Node

As mentioned before, `Node` has multiple subclasses, which are used to save the information of BPEL activities. In the implementation, instance of `Node` and its various subclass will be used. In order to manipulate the instance of `Node` and all kinds of its subclasses easier, a *static* class `NodeFactory` (cf. figure 3.4) is defined.

For example, instead of the constructed function of the various classes the method GETNODE (cf. figure 3.5) is used to get the instance of the classes. After parsing, the tags' name [BPSM+97] of the elements in BPEL file can be gotten as a QName [BHL+06]. E.g. the QName of a XML element `<sequence />` is `sequence`. Use the constructed function of the subclass `Sequence`, the instance of the XML element `<sequence />` can be initialized. But in the implementation what we got is the QName instead of the value of the QName. In order to get the instance of according classes, the method GETNODE is used in the following implementation. For the same reason, the method GENERATEGRAMMAR is specified for generating grammar.

In this class, the constants which are used in Node and its subclasses are defined as data members. It's easier to maintain them. There are three types of the constants:

- Element names of BPEL document: saves the used tags' QName of XML elements.
- Element attributes of BPEL document: saves the used name of tags' attributes of XML elements.
- Types of non-terminal: saves the used types' name of non-terminal symbols of formal grammar.

4. Implementation of Generating Grammar

After parsing BPEL file, all the needed information of BPEL process are saved in a tree model, `Node`. This chapter focuses on the implementation of generating grammar for BPEL process. The approach to transform structured activities of BPEL to formal grammar is represented in [Goe13]. To get the grammar for entire process, the approach of traversing the `Node`, and merging of grammar are introduced. The services, which are called in the process, need to specified. And the implementation should be tested and verified.

In this chapter the architecture of implementation is introduced in section 4.1. The section 4.2 shows, how the `Node` of a BPEL process is traversed, and the grammars of each activities of the process are merged together for the entire process. In section 4.3 uses two test cases to verify the implementation of this work.

4.1. Architecture

As figure 4.2 shows, the implementation in this work is consisted of three packages, `unifiedBpel`, `unifiedBpel.io` and `node2grammar`. The related package `Grammar` is a predefined package (cf. figure A.1), not the part of this work. The class `Project` in package `unifiedBpel` is the entrance of the whole process. The file name of the BPEL process is saved in its data member `filename` as the input. Then it can be executed by using the procedure RUN. It's comprised of three steps:

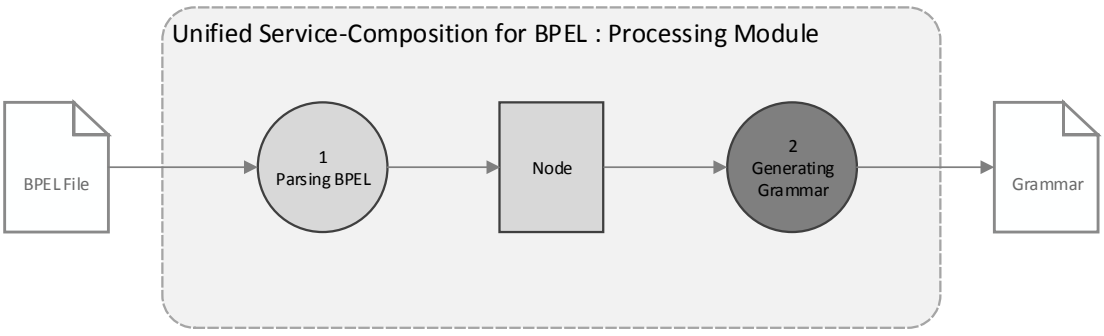


Figure 4.1.: The Overview of the Processing Module with Generating Grammar (Step 2) emphasized

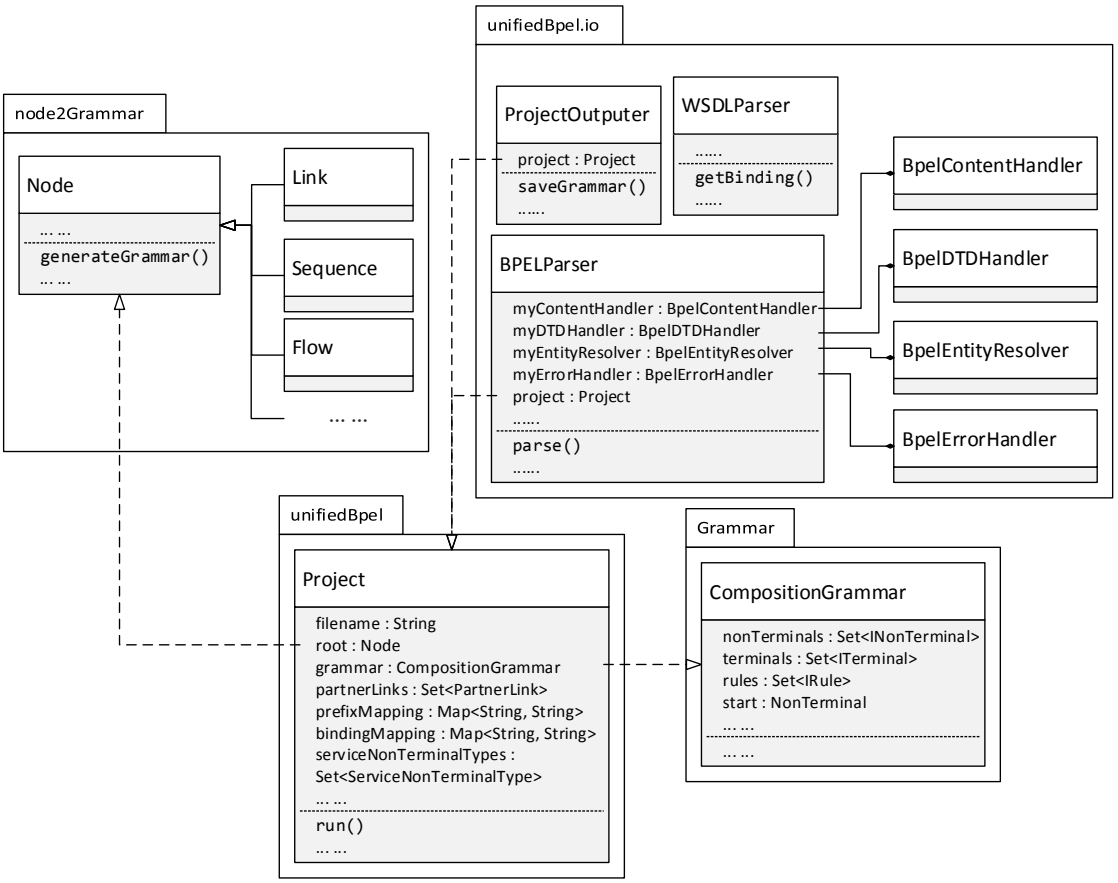


Figure 4.2.: Class Diagram for Architecture of Implementation

1. Parsing BPEL: as mentioned before, the parsing BPEL file is introduced. It is executed in this step, all the information of the BPEL process are saved in the variable `root`, which defined as type `Node`.
2. Generating Grammar: based on the work of [Goe13], the grammar of the BPEL process is generated here, and saved in variable `grammar`.
3. Outputting Grammar: in previous step generated formal grammar is saved to a XML document in this step.

The functionality of first step and third step are implemented in package `unifiedBpel.io`. The functionality of second step is implemented in package `node2Grammar`.

As mentioned before, the web services, which related to the BPEL process, and its calls should be specified. The service calls was introduced in [Goe13]. The following information of Web service are needed for the specification of service calls:

Listing 4.1 Syntax of `partnerLinks` [AAA+07]

```
1 <partnerLinks>
2    <partnerLink name="NCName"
3       partnerLinkType="QName"
4       myRole="NCName"?
5       partnerRole="NCName"?
6       initializePartnerRole="yes|no"? />+
7 </partnerLinks>
```

- Address: Address is web service endpoint[1], it is specified by using schema `/wsa:EndpointReference/wsa:Address`[2]. Its type should be `xs:anyURI`[3].
- operation: It's possible, multiple operation in a web service specified. It's should be specified, which operation in the service is used.
- binding: the binding of a web service is specified in its WSDL file. The binding information can be gotten by using class `WSDLParser` (cf. figure 4.2).
- portType: connection point of the service.

The address of a service should be acquired firstly. Because the binding information of the service need to be retrieved by access its WSDL document. In order to get the address value, the following steps are used:

1. Get the attribute `partnerLink`'s value of the activity.
2. Get the `partnerLinkType` by using `partnerLink`'s value. In class `Project` defines a variable `partnerLinks`, which is initialized after parsing BPEL. It saves all the specified `partnerLinks` in the BPEL file. As Listing 4.1 shows, the attribute `partnerLinkType` must be specified, i.e. by using `partnerLinks` can get the `partnerLinkType`.
3. The prefix of `partnerLinkType` is corresponding to a namespace, which the WSDL document located. The variable `prefixMapping` of class `Project` saves all the Mapping of prefix and namespaces in the BPEL file. The address of the WSDL is found.

Afterward the class `WSDLParser` parses the remote WSDL file, and return the binding's value, which is implemented in function getBinding. The variable `bindingMapping` of `Project` is used to save the gotten binding's value according to the service's address. The specified services are saved in variable `serviceNonTerminalTypes`. Once a service is to be specified, it should be checked, in order to avoid the reduplicated specification for the same service. There are four types of operation [CCM+01]. To call a specified service, the parameters are needed.

[1] A web service endpoint is a (referenceable) entity, processor, or resource where web service messages can be targeted. [BCC+04]
[2] wsa is the prefix of namespace http://schemas.xmlsoap.org/ws/2004/08/addressing.
[3] xs is the prefix of namespace http://www.w3.org/2001/XMLSchema

Listing 4.2 Syntax of `invoke` [AAA+07]

```
<invoke partnerLink="NCName"
   portType="QName"?
   operation="NCName"
   inputVariable="BPELVariableName"?
   outputVariable="BPELVariableName"?
   standard-attributes>
   ...
</invoke>
```

During the process there are some data generated and needed. The purpose of class `Project` is to organize them. For example, a `<invoke>` activity calls a remote service, based on [Goe13], the information of the should be defined as type `ServiceNonTerminalType`. As Listing 4.2 shows, there is no `address` attribute. In order to get it, with the help of the variable `partnerLinks` can get the prefix of the `partnerLinkType`, and with the help of `prefixMapping` can get the datum of address.

The package `unifiedBpel.io` means **I**nput and **O**utput functionality for the package `unifiedBpel`, it focuses on reading and writing document for `unifiedBpel`. In this package class `BPELParser` is defined to implement the input functionality. As mentioned before, SAX was chosen as parsing method. Therefore four classes are defined as handlers of SAX, `BpelContentHandler`, `BpelDTDHandler`, `BpelEntityResolver`, `BpelErrorHandler`. By using those classes the functionality of class `BPELParser` can be realized. Class `WSDLParser` is used to get the binding information of service calls. The output functionality is implemented in class `ProjectOutputer`. SAX can not modify XML document, therefore `ProjectOutputer` uses DOM to realize the functionality. There are no direct relations between the input classes and class `ProjectOutputer`, `WSDLParser`.

The functionality of second step of the function RUN is realized in package `node2Grammar`, i.e. in this package `node2Grammar` the BPEL process is transformed to formal grammar. The package `node2Grammar` is consisted of class `Node` and its subclass, and the utility class of class `NodeFactory`. As mentioned before, after parsing the needed information were saved in intermediate model, `Node`. And the BPEL process is reconstructed as a tree model. In order to get the formal grammar of the whole BPEL process, each element of the tree model should be traversed. In this work the depth-first algorithm is used for traversing.

4.2. Transformation

The root element of a BPEL file is `<process>`. A particular BPEL process can be composed by various activities. Each activities except the root can be nested by another one. As a type of XML document, activities of a BPEL process are constructed as a tree model. As mentioned before, after parsing the activities of BPEL file are saved into a tree model, `Node`, which has the same structure of the original BPEL process. In order to unify the process to formal grammar, the tree model need to be traversed.

Listing 4.3 Example to Explain Merging Grammars of BPEL Activities

```
1 <sequence>
2    <flow>
3       <invoke name="S1" />
4       <invoke name="S2" />
5    </flow>
6    <invoke name="S3" />
7 </sequence>
```

There are two common traversing algorithm, depth-first and breath-first. In structured activities of BPEL, its subsequent activity can not be activated, before the end of its execution. Listing 4.3 shows a `<flow>` activity and its sibling activity, `<invoke name="`S_3`" />`. Based on the specification of BPEL, if the child `<invoke>` S_1 activated and `<invoke>` S_2 unactivated, means the execution of `<flow>` is not finished. In this case, the `<invoke>` can not be activated, because the execution of `<flow>` isn't finished yet. In general case, if a BPEL activity has child activities, based on its specification its children are dealt with before the end of its execution. Therefore to transform the behavior of BPEL process to the same behavior, which is described by the corresponded grammar, depth-first algorithm is used for traversing in this work.

In the work of [Goe13], introduced the approach of transformation for structured activities of BPEL. To get the grammar of entire BPEL process, the grammar of each activity of the process should be merged together. In BPEL process at the end of the execution of a activities, based on the specification its subsequent activity should be activated by it. As mentioned before, the production rules, which produce terminal states of the activity, are saved in `FinalRules`. Note that BPEL activity may have more than one end state, and it's also possible, multiple production rules produces the same state. By using the help of `FinalRules` of a activity, its subsequent activity can be activated. Based on the approach of [Goe13], the grammar of `<flow>` in Listing 4.3 has the following production rules:

$$
\begin{array}{lrl}
(1) & Start & \longrightarrow S_1 S_2 H \\
(2) & S_1 & \longrightarrow s_1 \\
(3) & S_2 & \longrightarrow s_2 \\
(4) & s_1 s_2 H & \longrightarrow s_1 s_2 \\
(5) & s_2 s_1 H & \longrightarrow s_2 s_1
\end{array}
$$

$$
\begin{array}{ll}
with & S_i \in V \\
 & s_i \in \Sigma.
\end{array}
$$

Its `FinalRules` are rule (4-5). Both of them mean the execution of `<flow>` is finished. The different is, rule (4) means S_1 finished earlier than S_2, whereas rule (5) means S_2 finished earlier than S_1. As its subsequent activity, the `<invoke name="`S_3`" />` should be activated at the end of execution of the `<flow>`, i.e. both of rule (4) and rule (5) need to activate `invoke` S_3. Its `FinalRules` (4-5) should be modified as the following:

$$
\begin{aligned}
(4) \quad s_1 s_2 H &\longrightarrow s_1 s_2 S_3 \\
(5) \quad s_2 s_1 H &\longrightarrow s_2 s_1 S_3 \\
(6) \quad S_3 &\longrightarrow s_3
\end{aligned}
$$

The start symbol of the `<invoke name="`S_3`" />`'s grammar, S_3 was added into the end of right-hand-side (*rhs*) of the rules (4-5). The meaning of the new rules (4-5) is after finish of execution of the `<flow>`, the S_3 is activated. With the new rules (4-5), rules (1-6) mean the merging of the grammar of the `<flow>` and `<invoke name="`S_3`" />`.

Note that the subsequent activity of a activity in BPEL process means behavior subsequence. It's not decided by itself, but by its parent activity. In the example of listing 4.3, the parent activity of the `<flow>` and `<invoke name="`S_3`" />` is a `<sequence>`. Based on the specification of `<sequence>` activity in BPEL, the `<invoke name="`S_3`" />` is the subsequent activity of the `<flow>`. If the `<sequence>` is changed to a `<flow>`, the `<invoke name="`S_3`" />` is no longer the subsequent activity of the `<flow>`. Both of them should be executed parallel. Therefore the merging of grammars for activities should be specified in its parent activity. Every structured activities decides the behavior relations of its children activities, which specified in [AAA+07]. The merging of grammar in different scene is not the same. Based on the behavior relations of activities, which are specified in parent activity, the merging of grammar is different. Consequently the method GENERATEGRAMMAR of each subclasses of `Node` has different algorithm, to generate grammar of activities.

4.3. Test Cases

In this section, two test cases are used, to verify the implementation of this work. The first test cases in this section is example from [AAA+07]. In this test case the basic activities and structured activities are covered. The second test case is from [Goe13], it tests the transformation of `<scope>`. To understand the two processes easier they are both represented graphical. To verify the results, the production rules based on the approach of [Goe13] and generated by implementation are compared.

Auction Service

This test case uses a example from [AAA+07, Section 15.4]. This example describes a simplified auction house process. At the beginning of the process, the needed variables are defined. As figure 4.3 shows, at first step of the process two `<receive>` activities with name `acceptSellerInformation` and `acceptBuyerInformation` are defined to collect information of seller and buyer of a auction. The two `<receive>` has no context relations, therefore they are contained by a `<flow>`. After collection the information are saved to the predefined variables by using the `<assign>`. The activity `registerAuctionResults` is a `<invoke>`. It calls a service to register the auction, i.e. a instance of auction is initialized. As a `<receive>`

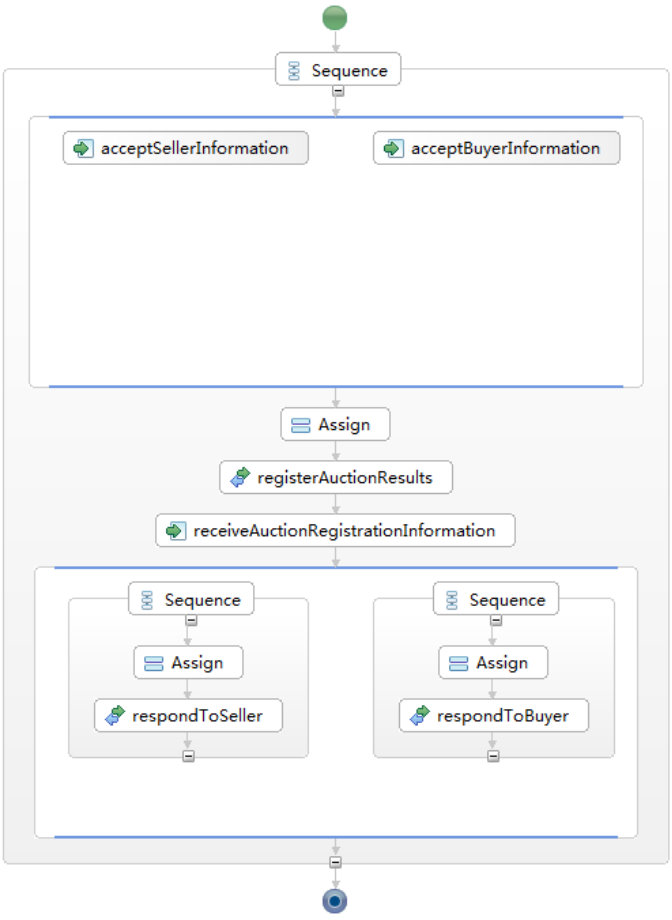


Figure 4.3.: Activity Diagram of the Process Auction Service.

`receiveAuctionRegistrationInformation` collects the information of the registered auction.

A particular auction is actually a information interactive system. It collects the needed information, and based on its business logic, makes a decision. At the end, the result should be responded to the seller and buyer. In this process, what in the final step does, is sending responses back to seller and buyer. There are two `<sequence>`s, one for seller, another for buyer. The corresponded `<assign>` collects the information, which need to be sent. The two `<invoke>`, `respondToSeller` and `respondToBuyer` do the responses. The two `<sequence>`s has no relations, therefore they are contained by a `<flow>`. The ordering of the steps is firmly. The information of seller, buyer and auction should be collected firstly. Based on the logic process the information. And respond the result back. Therefore those steps of entire process is bounded in a `<sequence>`.

Variables :	$D_1 ... D_6 \longleftrightarrow VARIABLE7 ... VARIABLE12$
First `<flow>` :	$S_1 \longleftrightarrow FLOW16, H_1 \longleftrightarrow FLOW16.SYNC$
`acceptSellerInformation` :	$S_2 \longleftrightarrow RECEIVE17, s_2 \longleftrightarrow receive17$
`acceptBuyerInformation` :	$S_3 \longleftrightarrow RECEIVE20, s_3 \longleftrightarrow receive20$
`<assign>` after first `<flow>` :	$D_7 ... D_10 \longleftrightarrow COPY24 ... COPY38, d_7 ... d_10 \longleftrightarrow copy24 ... copy38$
`registerAuctionResults` :	$S_4 \longleftrightarrow INVOKE41, s_4 \longleftrightarrow invoke41$
`receiveAuctionRegistrationInformation` :	$S_5 \longleftrightarrow RECEIVE42, s_5 \longleftrightarrow receive42$
Second `<flow>` :	$S_6 \longleftrightarrow FLOW45, H_2 \longleftrightarrow FLOW45.SYNC$
`<assign>` for seller responses :	$D_11, D_12 \longleftrightarrow COPY48, COPY51, d_11, d_12 \longleftrightarrow copy48, copy51$
`respondToSeller` :	$S_7 \longleftrightarrow INVOKE55, s_7 \longleftrightarrow invoke55$
`<assign>` for buyer responses :	$D_13, D_14 \longleftrightarrow COPY58, COPY61, d_13, d_14 \longleftrightarrow copy58, copy61$
`respondToBuyer` :	$S_8 \longleftrightarrow INVOKE65, s_8 \longleftrightarrow invoke65$
with	Symbols in 4.5(a) $\longleftrightarrow$ Symbols in 4.5(b)

Figure 4.4.: The Relation of Symbols, which are used in 4.5(a) and 4.5(b)

Based on the the approach of [Goe13] and merging of grammar. The production rules of the process is presented in figure 4.5(a), and the rules from the implementation are shows in figure 4.5(b). In order to observe them easier, the figure 4.4 shows the relations of non-terminals and terminals between them. There are six variables defined in this process, which represented by rules (1-7) of figure 4.5(a) and figure 4.5(b). The rules (8-12) of figure 4.5(a) and figure 4.5(b) mean the first step of the process, collect the information of seller and buyer. After synchronization rules (12-13) the `<assign>` is activated, which is shown by rules (13-16). The service calls for `registerAuctionResults` and `receiveAuctionRegistrationInformation` are represented by rules (17-18). In the last step, functionality of the `<assign>` and `respondToSeller` is shown in rules (20-22), rules (23-25) is for another `<assign>` and `respondToBuyer`. The two group of rules are activated by the last `<flow>` activity, which rule (19) shows. The synchronization of the `<flow>` is represented by rules (26-27). By using the help of figure 4.4 it can be concluded, the production rules of figure 4.5(a) and figure 4.5(b) represents the same process.

$$
\begin{array}{lrl}
(1) & Start & \longrightarrow D_1 \\
(2) & D_1 & \longrightarrow D_2 \\
(3) & D_2 & \longrightarrow D_3 \\
(4) & D_3 & \longrightarrow D_4 \\
(5) & D_4 & \longrightarrow D_5 \\
(6) & D_5 & \longrightarrow D_6 \\
(7) & D_6 & \longrightarrow S_1 \\
(8) & S_1 & \longrightarrow S_2 S_3 H_1 \\
(9) & S_2 & \longrightarrow s_2 \\
(10) & S_3 & \longrightarrow s_3 \\
(11) & s_2 s_3 H_1 & \longrightarrow s_2 s_3 D_7 \\
(12) & s_3 s_2 H_1 & \longrightarrow s_3 s_2 D_7 \\
(13) & D_7 & \longrightarrow d_7 D_8 \\
(14) & D_8 & \longrightarrow d_8 D_9 \\
(15) & D_9 & \longrightarrow d_9 D_{10} \\
(16) & D_{10} & \longrightarrow d_{10} S_4 \\
(17) & S_4 & \longrightarrow s_4 S_5 \\
(18) & S_5 & \longrightarrow s_5 S_6 \\
(19) & S_6 & \longrightarrow D_{11} D_{13} H_2 \\
(20) & D_{11} & \longrightarrow d_{11} D_{12} \\
(21) & D_{12} & \longrightarrow d_{12} S_7 \\
(22) & S_7 & \longrightarrow s_7 \\
(23) & D_{13} & \longrightarrow d_{13} d_{14} \\
(24) & D_{14} & \longrightarrow d_{14} S_8 \\
(25) & S_8 & \longrightarrow s_8 \\
(26) & s_7 s_8 H_2 & \longrightarrow s_7 s_8 \\
(27) & s_8 s_7 H_2 & \longrightarrow s_8 s_7
\end{array}
$$

$$
\begin{array}{ll}
with & S_i \in Services \\
& D_i \in DataContainer \\
& H_i \in Helpers \\
& s_i \in \Sigma.
\end{array}
$$

(a) Production rules of auction service based on [Goe13]

$$
\begin{array}{lrl}
(1) & Start & \longrightarrow VARIABLE7 \\
(2) & VARIABLE7 & \longrightarrow VARIABLE8 \\
(3) & VARIABLE8 & \longrightarrow VARIABLE9 \\
(4) & VARIABLE9 & \longrightarrow VARIABLE10 \\
(5) & VARIABLE10 & \longrightarrow VARIABLE11 \\
(6) & VARIABLE11 & \longrightarrow VARIABLE12 \\
(7) & VARIABLE12 & \longrightarrow FLOW16 \\
(8) & FLOW16 & \longrightarrow RECEIVE17\ RECEIVE20\ FLOW16.SYNC \\
(9) & RECEIVE17 & \longrightarrow receive17 \\
(10) & RECEIVE20 & \longrightarrow receive20 \\
(11) & receive17\ receive20\ FLOW16.SYNC & \longrightarrow receive17\ receive20\ COPY24 \\
(12) & receive20\ receive17\ FLOW16.SYNC & \longrightarrow receive20\ receive17\ COPY24 \\
(13) & COPY24 & \longrightarrow copy24\ COPY32 \\
(14) & COPY32 & \longrightarrow copy32\ COPY35 \\
(15) & COPY35 & \longrightarrow copy35\ COPY38 \\
(16) & COPY38 & \longrightarrow copy38\ INVOKE41 \\
(17) & INVOKE41 & \longrightarrow invoke41\ RECEIVE42 \\
(18) & RECEIVE42 & \longrightarrow receive42\ FLOW45 \\
(19) & FLOW45 & \longrightarrow COPY48\ COPY58\ FLOW45.SYNC \\
(20) & COPY48 & \longrightarrow copy48\ COPY51 \\
(21) & COPY51 & \longrightarrow copy51\ INVOKE55 \\
(22) & INVOKE55 & \longrightarrow invoke55 \\
(23) & COPY58 & \longrightarrow copy58\ COPY61 \\
(24) & COPY61 & \longrightarrow copy61\ INVOKE65 \\
(25) & INVOKE65 & \longrightarrow invoke65 \\
(26) & invoke55\ invoke65\ FLOW45.SYNC & \longrightarrow invoke55\ invoke65 \\
(27) & invoke65\ invoke55\ FLOW45.SYNC & \longrightarrow invoke65\ invoke55
\end{array}
$$

(b) Production rules of auction service from the implementation

Figure 4.5.: Compare of the Production Rules of Auction Service based on [Goe13] and generated by Implementation

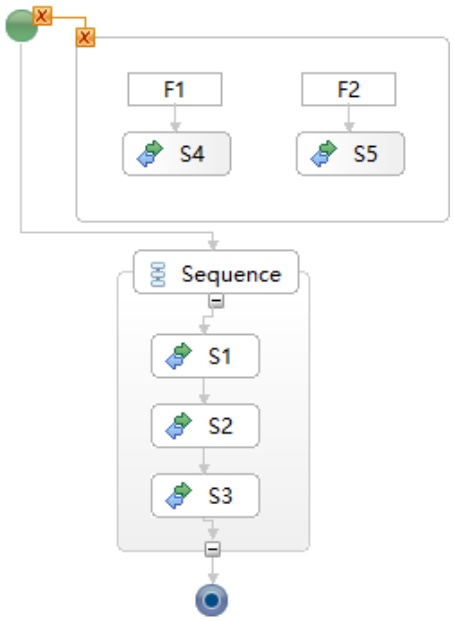


Figure 4.6.: Activity Diagram of the Process Scope with user-defined faultHandlers

Scope with user-defined faultHandlers

This test case uses a example from [Goe13, Figure 17], which was used to describe the transformation for a `<scope>` with user-defined fault handlers. As a reference it is added in the appendix of this work. The source code of the process is shown in A.2(a). This process is very simple. As figure 4.6 shows, a `<scope>` has a sequence of three `<invoke>` activities with name S_1, S_2, S_3, and two fault handlers F_1, F_2. F_1 has a `<invoke>` activity S_4, which will be activated, once F_1 is triggered. Similarly <invoke> S_5 is for F_2.

The production rules based on approach of [Goe13] are shown in [Goe13, Figure 17(b)], it can be also located in figure A.2(b). Figure 4.8 shows the production rules from the implementation. To observe them easier, the relations of the symbols, which used in grammar, is summarized in figure 4.7. In the process two fault handlers are for `<scope>` defined, which means there are three state of execution, regular run and two fault handling mode. There are indicated by t_1, f_1 and f_2 in A.2(b). Rules (2-9) and (11-12) represents the behavior of the $Fault_1$, similarly rules (13-22) is for $Fault_2$. Rules (23-33) shows the execution of S_1, S_2, S_3. Before execute them, the fault handlers should be activated first, which is represented by the right-hand-side of rules (1,23,26). If the state is t_1, S_1, S_2, S_3 can be executed. If the state is f_1 or f_2, they are aborted, which is shown in rules (24-25, 27-28, 30-31). Rules (34-37) adapt the execution state to the end state. With the help of figure 4.7, it can be verified, the production rules of [Goe13, Figure 17(b)] and 4.8 are accordant.

`<scope>` :	$R_1 \longleftrightarrow START.R, T_1 \longleftrightarrow START.T, t_1 \longleftrightarrow start.t, H'_8 \longleftrightarrow START.FAULT'$
:	$b \longleftrightarrow start.b, r \longleftrightarrow start.r, q \longleftrightarrow start.q$
`<invoke>`s :	$S_1 ... S_3 \longleftrightarrow INVOKE7 ... INVOKE9, S_4 \longleftrightarrow INVOKE3, S_5 \longleftrightarrow INVOKE5$
:	$s_1 ... s_3 \longleftrightarrow invoke7 ... invoke9, s_4 \longleftrightarrow invoke3, s_5 \longleftrightarrow invoke5$
Fault handler F_1 :	$H_1 \longleftrightarrow CATCH2.FAULT, H'_1 \longleftrightarrow CATCH2.FAULT', H_3 \longleftrightarrow CATCH2.H$
:	$F_1 \longleftrightarrow CATCH2.F, f_1 \longleftrightarrow catch2.f, b'_1 \longleftrightarrow catch2.b$
Fault handler F_2 :	$H_2 \longleftrightarrow CATCH4.FAULT, H'_2 \longleftrightarrow CATCH4.FAULT', H_4 \longleftrightarrow CATCH4.H$
:	$F_2 \longleftrightarrow CATCH4.F, f_2 \longleftrightarrow catch2.f, b''_1 \longleftrightarrow catch4.b$
with	Symbols in [Goe13, Figure 17(b)] $\longleftrightarrow$ Symbols in 4.8

Figure 4.7.: The Relation of symbols, which are used in [Goe13, Figure 17(b)] and 4.8

(1)	$Start$	$\longrightarrow start.t\ CATCH2.FAULT\ CATCH4.FAULT\ INVOKE7\ START.R$
(2)	$start.t\ CATCH2.FAULT$	$\longrightarrow START.T$
(3)	$start.t\ CATCH2.FAULT$	$\longrightarrow CATCH2.F$
(4)	$catch2.f\ CATCH2.FAULT$	$\longrightarrow catch2.f$
(5)	$catch4.f\ CATCH2.FAULT$	$\longrightarrow catch4.f$
(6)	$start.t\ CATCH2.FAULT'$	$\longrightarrow START.T\ catch2.b$
(7)	$start.t\ CATCH2.FAULT'$	$\longrightarrow CATCH2.F\ catch2.b$
(8)	$catch2.f\ CATCH2.FAULT'$	$\longrightarrow catch2.f\ catch2.b$
(9)	$catch4.f\ CATCH2.FAULT'$	$\longrightarrow catch4.f\ catch2.b$
(10)	$START.T$	$\longrightarrow start.t$
(11)	$CATCH2.F$	$\longrightarrow catch2.f\ CATCH2.H$
(12)	$CATCH2.H$	$\longrightarrow INVOKE3$
(13)	$start.t\ CATCH4.FAULT$	$\longrightarrow START.T$
(14)	$start.t\ CATCH4.FAULT$	$\longrightarrow CATCH4.F$
(15)	$catch2.f\ CATCH4.FAULT$	$\longrightarrow catch2.f$
(16)	$catch4.f\ CATCH4.FAULT$	$\longrightarrow catch4.f$
(17)	$start.t\ CATCH4.FAULT'$	$\longrightarrow START.T\ catch4.b$
(18)	$start.t\ CATCH4.FAULT'$	$\longrightarrow CATCH4.F\ catch4.b$
(19)	$catch2.f\ CATCH4.FAULT'$	$\longrightarrow catch2.f\ catch4.b$
(20)	$catch4.f\ CATCH4.FAULT'$	$\longrightarrow catch4.f\ catch4.b$
(21)	$CATCH4.F$	$\longrightarrow catch4.f\ CATCH4.H$
(22)	$CATCH4.H$	$\longrightarrow INVOKE5$
(23)	$start.t\ INVOKE7$	$\longrightarrow start.t\ invoke7\ CATCH2.FAULT\ CATCH4.FAULT\ INVOKE8$
(24)	$catch2.f\ INVOKE7$	$\longrightarrow catch2.f$
(25)	$catch4.f\ INVOKE7$	$\longrightarrow catch4.f$
(26)	$start.t\ INVOKE8$	$\longrightarrow start.t\ invoke8\ CATCH2.FAULT\ CATCH4.FAULT\ INVOKE9$
(27)	$catch2.f\ INVOKE8$	$\longrightarrow catch2.f$
(28)	$catch4.f\ INVOKE8$	$\longrightarrow catch4.f$
(29)	$start.t\ INVOKE9$	$\longrightarrow start.t\ invoke9\ CATCH2.FAULT'\ CATCH4.FAULT'\ START.FAULT'$
(30)	$catch2.f\ INVOKE9$	$\longrightarrow catch2.f$
(31)	$catch4.f\ INVOKE9$	$\longrightarrow catch4.f$
(32)	$INVOKE3$	$\longrightarrow invoke3$
(33)	$INVOKE5$	$\longrightarrow invoke5$
(34)	$catch2.b\ catch4.b\ START.FAULT'$	$\longrightarrow start.b$
(35)	$start.t\ start.b\ START.R$	$\longrightarrow start.r$
(36)	$catch2.f\ start.b\ START.R$	$\longrightarrow start.q$
(37)	$catch4.f\ start.b\ START.R$	$\longrightarrow start.q$

Figure 4.8.: Production Rules of [Goe13, Figure 17(a)] generated by Implementation

5. Summary

The work in this thesis focuses on unified service composition for BPEL. Based on the concept of [Goe13], with the help of formal grammar BPEL process is transformed to a unified model. In this work the first phase is parsing BPEL process, which is described in chapter 3. The class `Node` is defined as a intermediate model in section 3.1 to save the information of activities for BPEL process. In order to get more efficiency of execution, SAX is used to read and parse BPEL process (cf. section 3.2). A utility class `NodeFactory` is defined in section 3.3, to manipulate `Node` and its subclass easier. After parsing the `Node`, which saves the information of BPEL process, is constructed to the same structure of original process.

The second phase, generating grammar, which is represented in chapter 4, is focuses to get grammar from entire BPEL process. The architecture of implementation is introduced in section 4.1. Three packages are used, `unifiedBpel`, `unifiedBpel.io` and `node2Grammar`. The class `Project` in package `unifiedBpel` is the processing's entrance of this work. The approach to specify web service and its calls is explained and implemented. The classes in package `unifiedBpel.io` handle the inputting and outputting of the work, e.g. parsing BPEL process. The class `Node` and its subclass, which is introduced in chapter 3, are bounded in package `node2Grammar`. It's functionality is generating grammar. Based on the behavior of BPEL activities, in each subclass of `Node` the method GENERATEGRAMMAR is implemented to generate its grammar. The section 4.2 introduces the traversing of the tree model, which is outputted by parsing. The depth-first algorithm is used to traverse the tree. To merge production rules of grammars the principle of merging is also explained. Two test cases are used to test and verify the implementation in section 4.3.

A. Appendix

A.1. Predefined Package Grammar

In this work the BPEL process is transformed to formal grammar. The predefined package `grammar` is used in multiple places. In this package defines a data structure for grammar, class `CompositionGrammar`. As figure A.1 shows, its data members, `start`, `nonTeminals`, `terminals`, `rules` are defined as the tuple's elements of formal grammar. Class `NonTerminal` is specified for non-terminal symbols, class `Terminal` for terminal symbols, class `Rule` for production rules. Note that, the class `ServiceNonTerminal` is specified for the non-terminal symbols, who need call services.

A.2. Test case: Scope with user-defined faultHandlers

Figure A.2 shows the transformation for user-defined fault handlers in scope from [Goe13]. In this work it is used as a test case, to verify the production rules of the implemented grammar.

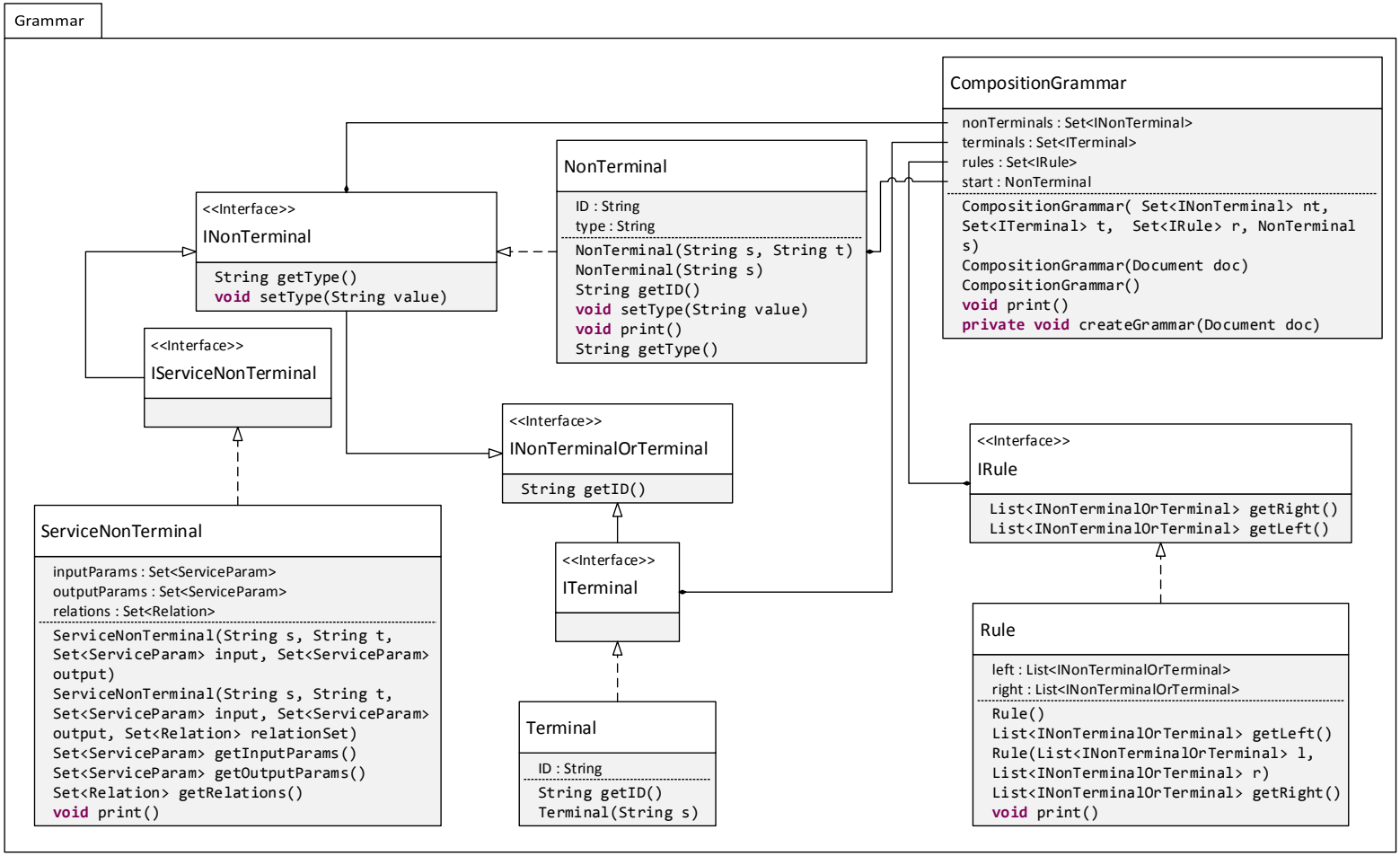


Figure A.1.: The Classes in Package Grammar and their Relations

```
<scope name="R">
<faultHandlers>
<catch faultname="Fault_1">
<invoke name="S_4" />
</catch>
<catch faultname="Fault_1">
<invoke name="S_5" />
</catch>
<faultHandlers>
<sequence>
<invoke name="S_1" />
<invoke name="S_2" />
<invoke name="S_3" />
</sequence>
</scope>
```

(a) BPEL specification of a scope with user-defined fault handlers.

$$
\begin{aligned}
(1)\quad & Start && \longrightarrow t_1 H_1 H_2 S_1 R_1 \\
(2)\quad & t_1 H_1 && \longrightarrow T_1 \\
(3)\quad & t_1 H_1 && \longrightarrow F_1 \\
(4)\quad & f_1 H_1 && \longrightarrow f_1 \\
(5)\quad & f_2 H_1 && \longrightarrow f_2 \\
(6)\quad & t_1 H_1' && \longrightarrow T_1 b_1' \\
(7)\quad & t_1 H_1' && \longrightarrow F_1 b_1' \\
(8)\quad & f_1 H_1' && \longrightarrow f_1 b_1' \\
(9)\quad & f_2 H_1' && \longrightarrow f_2 b_1' \\
(10)\quad & T_1 && \longrightarrow t_1 \\
(11)\quad & F_1 && \longrightarrow f_1 H_3 \\
(12)\quad & H_3 && \longrightarrow S_4 \\
(13)\quad & t_1 H_2 && \longrightarrow T_1 \\
(14)\quad & t_1 H_2 && \longrightarrow F_2 \\
(15)\quad & f_1 H_2 && \longrightarrow f_1 \\
(16)\quad & f_2 H_2 && \longrightarrow f_2 \\
(17)\quad & t_1 H_2' && \longrightarrow T_1 b_1'' \\
(18)\quad & t_1 H_2' && \longrightarrow F_2 b_1'' \\
(19)\quad & f_1 H_2' && \longrightarrow f_1 b_1'' \\
(20)\quad & f_2 H_2' && \longrightarrow f_2 b_1'' \\
(21)\quad & F_2 && \longrightarrow f_2 H_4 \\
(22)\quad & H_4 && \longrightarrow S_5 \\
(23)\quad & t_1 S_1 && \longrightarrow t_1 s_1 H_1 H_2 S_2 \\
(24)\quad & f_1 S_1 && \longrightarrow f_1 \\
(25)\quad & f_2 S_1 && \longrightarrow f_2 \\
(26)\quad & t_1 S_2 && \longrightarrow t_1 s_2 H_1 H_2 S_3 \\
(27)\quad & f_1 S_2 && \longrightarrow f_1 \\
(28)\quad & f_2 S_2 && \longrightarrow f_2 \\
(29)\quad & t_1 S_3 && \longrightarrow t_1 s_3 H_1' H_2' H_8' \\
(30)\quad & f_1 S_3 && \longrightarrow f_1 \\
(31)\quad & f_2 S_3 && \longrightarrow f_2 \\
(32)\quad & S_4 && \longrightarrow s_3 \\
(33)\quad & S_5 && \longrightarrow s_5 \\
(34)\quad & b_1' b_1'' H_8' && \longrightarrow b_1 \\
(35)\quad & t_1 b_1 R_1 && \longrightarrow r_1 \\
(36)\quad & f_1 b_1 R_1 && \longrightarrow q_1 \\
(37)\quad & f_2 b_1 R_1 && \longrightarrow q_1
\end{aligned}
$$

$$
\begin{aligned}
with: \quad S_i &\in V \\
H_1, H_2 &\in Faults \\
R_x, H_3, H_4, T_1, F_y &\in Helpers \\
f_1, f_2, s_i, t_1, b_1, b_1', b_1'' &\in \Sigma
\end{aligned}
$$

(b) Production rules for the user-defined fault handlers H_1 and H_2 in a scope.

Figure A.2.: Representation of user-defined fault handlers. [Goe13]

Bibliography

[AAA+07] A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guzar, N. Kartha, et al. Web services business process execution language version 2.0 (OASIS standard). WS-BPEL TC OASIS, 2007. (Cited on pages 6, 7, 9, 15, 21, 22 and 24)

[AHW03] W. Van der Aalst, A. ter Hofstede, M. Weske. Business process management: A survey. *Business Process Management*, pp. 1019–1019, 2003. (Cited on page 9)

[BCC+04] D. Box, E. Christensen, F. Curbera, D. Ferguson, J. Frey, M. Hadley, C. Kaler, D. Langworthy, F. Leymann, B. Lovering, et al. Web services addressing (WS-Addressing), 2004. (Cited on pages 10 and 21)

[BHL+06] T. Bray, D. Hollander, A. Layman, R. Tobin, H. S. Thompson. Namespaces in XML 1.0 . W3C recommendation. *World Wide Web Consortium (W3C)*, 2006. (Cited on page 17)

[BPSM+97] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau. Extensible markup language (XML). *World Wide Web Journal*, 2(4):27–66, 1997. (Cited on pages 10 and 17)

[C+01] W. W. W. Consortium, et al. Document Object Model (DOM), 2001. (Cited on page 16)

[CCM+01] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, et al. Web services description language (WSDL) 1.1, 2001. (Cited on pages 10 and 21)

[CD+99] J. Clark, S. DeRose, et al. XML path language (XPath) version 1.0, 1999. (Cited on page 10)

[Fer04] C. Ferris. Web services architecture. *Standard, W3C World*, 2004. (Cited on page 10)

[Goe13] K. Goerlach. A generic transformation of existing service composition models to a unified model. Technical report, Institute of Architecture of Application Systems, University of Stuttgart, 2013. (Cited on pages 6, 19, 20, 22, 23, 24, 26, 27, 28, 29, 31, 33 and 35)

[Gor07] O. Gorter. Java XML Parsing: SAX vs DOM. *URL: http://tech.inhelsinki.nl/2007-08-29/*, 2007. (Cited on pages 6, 15 and 16)

[M+98] D. Megginson, et al. Simple api for xml. *URL: http://www.saxproject.org*, 1998. (Cited on page 16)

[OGA+05] T. Oinn, M. Greenwood, M. Addis, M. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, et al. Taverna: lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, 2005. (Cited on page 7)

[PA06] M. Pesic, W. van der Aalst. A declarative approach for flexible business processes management. In *Business Process Management Workshops*, pp. 169–180. Springer, 2006. (Cited on page 7)

[WAV04] M. Weske, W. Van der Aalst, H. Verbeek. Advances in business process management. *Data and Knowledge Engineering*, 50(1):1–8, 2004. (Cited on pages 6 and 9)

All links were last followed on May 10, 2013.

Declaration

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

Ort, Datum, Unterschift