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

Studienarbeit Nr. 2175

Translating WS-BPEL 2.0 to BPELscript and Vice Versa

Marc Bischof

Course of Study:	Computer Science
Examiner:	Prof. Dr. F. Leymann
Supervisors:	Dipl.-Inf. O. Kopp Dipl.-Inf. T. van Lessen
Commenced:	May 1, 2008
Completed:	October 31, 2008
CR-Classification:	D.1.1, D.2.11, D.3.4, H.4.1, K.1

Contents

List of Abbreviations

ANTLR	ANother Tool for Language Recognition
ANTXR	ANother Tool for Xml Recognition
Apache ODE	Apache Orchestration Director Engine
API	Application Programming Interface
AST	Abstract Syntax Tree
BOSTO	BPEL to BPELscript Translator
BPEL	Business Process Execution Language
BPEL4WS	BPEL for Web Services
BPMN	Business Process Modeling Notation
DOM	Document Object Model
DSL	Domain Specific Language
GUI	Graphical User Interface
GUnit	Grammar Unit Testing
IMA	Inbound Message Activity
KISS	Keep It Simple and Stupid
LL-Recognizer	Recognize the input from **l**eft to right using the **l**eftmost derivation.
LR-Recognizer	Recognize the input from **l**eft to right using the **r**ightmost derivation.
MVA	Multi-Valued Attributes
OASIS	Organization for the Advancement of Structured Information Standards
PCDATA	Parsed Character Data
QName	Qualified Name
SAX	Simple API for XML
SimPEL	BPEL Simplified Syntax
SOA	Service Oriented Architecture
SSA	Static Single Assignment Form
WS-BPEL 2.0	Web Service Business Process Execution Language 2.0
WSFL	Web Services Flow Language
XLANG	XML LANGuage
XML	Extensible Markup Language
XPA	XML Processing for ANTLR
XSD	XML Schema Definition
XSLT	Extensible Stylesheet Language Transformations
YACC	Yet Another Compiler Compiler

Chapter 1

Introduction

The *Business Process Execution Language*—BPEL—is an XML-based (Extensible Markup Language) language to specify business processes with the intention to "act as the central controller of the business process" [BPEa]. It provides an interoperable and portable language for both abstract and executable processes. Thus, BPEL defines the orchestration of services within a SOA (Service Oriented Architecture) and is emerging as a standard way of defining business processes [WCL+05, p. 3].

BPEL is intended as a serialization format. Thus it is not a programming language at all and does not have a graphical representation. Mappings from graphical languages such as the Business Process Modeling Notation (BPMN [Obj08, WM08]) to BPEL are available ([Whi, Sch08, ODBt06]), but programmers familiar to syntax like Java, C, ... are disregarded. One option is to force the programmers to learn a completely new syntax. The other option is to introduce a new syntax to BPEL. To provide portability, it has to be possible to (a) map all valid programs expressed in the syntax to BPEL and (b) map all valid BPEL processes to programs expressed in the syntax.

In this work, the language *BPELscript* is presented, referring to the second option. *BPELscript* is a new syntax for BPEL and fulfills these two requirements.

1.1 BPEL

To gain a standardized way to describe business processes, IBM and Microsoft initiated BPEL4WS (BPEL for Web Services) as a new language which emerged finally as WS-BPEL 2.0 OASIS Standard [OAS07a] (Web Service Business Process Execution Language 2.0 of the Organization for the Advancement of Structured Information Standards). Thereby, BPEL combines IBMs graph-based WSFL (Web Services Flow Language) [Ley01], to describe the composition of Web Services, with Microsoft's calculus-based XLANG [Tha01]. To meet commonplace requirements in today's information technology, BPEL was designed from the beginning to operate in heterogeneity and dynamism. [KKL06] outlined the view of BPEL as a programming language in XML, an export format for business processes or a powerful workflow language that presents the natural (conversational) model for programming-in-the-large in a service oriented world. BPEL can be viewed as the glue, used to construct business processes with services (from a SOA system) as building blocks [Ley06].

1.2 Motivation—A Simplified Script Syntax for BPEL

As regarded before BPEL is a serialization format for which this work provides a simplified script syntax. For example, take the variable assignment of BPEL which is shown in Listing 1.1. As semi-structured XML, BPEL requires several lines of code overhead to specify a simple assignment activity. Nevertheless, it is possible to specify an assignment in a way which is easier to read and write like this is shown in the same listing.

Listing 1.1 Simple Variable Assignment in BPEL and Java

```
# BPEL assign
  <copy>
    <from>
       <literal>yes</literal>
    <from>
    <to part="accept" variable="approval"/>
  </copy>

# equivalent in Java
  approval.accept="yes";
```

To point out why object-oriented Java code is not the ideal solution, Listing 1.2 shows an example of BPELs Loan Approval Process in Java. At first glance, this looks better than original BPEL, but would it not be more familiar to replace the *public class Processor* against a *process loanApprovalProcess*?

Listing 1.2 Loan Approval in Java [BPEb]

```
public class Processor {
    public String assessLoanRequest(Request request) {
        Approval approval = new Approval();
        Risk risk = new Risk();
        if (request.amount < 10000) {
            risk.level = loanAssessor(request);
            if (risk.level == "low") {
                approval.accept="yes";
                return approval;
        }   }
        // Otherwise - unless we have returned
        risk.level = loanApprover(request);
        if (risk.level == "low") {
            approval.accept="yes";
            return approval;
}   }   }
```

Therefore the *Apache ODE-Group* (Apache Orchestration Director Engine [ODEb]) has recommended the *BPEL Simplified Syntax* called SimPEL [BAR, LK]. SimPEL is a Java Script based *Domain Specific Language* (DSL) with implicit variable declaration. The problem with SimPEL is, that programmers

who already knows BPEL has to cope with the scripting facilities of SimPEL. Furthermore, SimPEL misses the handling of BPELs standard elements. In contrast, this work proposes *BPELscript*, a possibility which provides a more graph-oriented scripting approach of SimPEL and stays close to BPEL by supporting all its facilities. Thereby, BPELscript mainly arises from two proposals of the ODE [BAR, LK] and the resulting language SimPEL [RE]. To give a first impression of *BPELscript*, Listing 1.3 illustrates the Loan Approval Process in *BPELscript*.

Listing 1.3 Example Loan Approval Process

```
namespace pns = "http://example.com/loan-approval/";
namespace lns = "http://example.com/loan-approval/wsdl/";
import lServicePT = lns::"loanServicePT.wsdl";

process pns::loanApprovalProcess {

partnerLink customer, approver, assessor;

try {
    parallel {
      request = receive(customer, request);
      signal(receive-to-assess, [$request.amount &lt; 10000]);
      signal(receive-to-approval, [$request.amount >= 10000]);
    } and {
      join(receive-to-assess);
      risk = invoke(assessor, check, request);
      signal(assess-to-setMessage, [$risk.level = 'low']);
      signal(assess-to-approval, [$risk.level != 'low']);
    } and {
      join(assess-to-setMessage);
      approval.accept = "yes";
      signal(setMessage-to-reply);
    } and {
      join(receive-to-approval, assess-to-approval);
      approval = invoke(approver, approve, request);
      signal(approval-to-reply);
    } and {
      join(approval-to-reply, setMessage-to-reply);
      reply(customer, request, approval);
    }
  } catch(lns::loanProcessFault) { |error|
    reply(customer, request, error);
  }
}
```

To gain in importance, a $1:1$ mapping from BPEL to BPELscript and vice versa is required. Therefore this work provides a translator using the ANTLR v3 (ANother Tool for Language Recognition) parser generator which is based on a predicated-LL(*) (refer to Section 2.1) parsing strategy. This decision was made to come up with an easy extensible framework of slippy changing prototypes.

This solution uses the implicit tree structure behind the input sentences to construct an *abstract syntax tree* (AST) which is a highly processed and condensed version of the input [Par07]. The translator maps each input sentence of the source language to an output sentence by embedding actions (e. g.

code) within the grammar or tree. This actions will be executed according to its position within the grammar or tree. To support an easy handling of implicit declarations the translation is broken down into multiple passes.

In the following, this work describes how BPELscript achieves its 1 : 1 mapping using automated tools to provide extensibility, maintainability and of course usability. Therefore the work is split into five chapters:

- **Chapter 2 "Introduction to Computer Language Translation and the Main Tools"** describes basics of language translation and introduces the main tools
- **Chapter 3 "BPELscript"** introduces the BPELscript syntax and discuss modeling alternatives and the mapping to BPEL
- **Chapter 4 "Project Description of *bosto*—The BPEL to BPELscript Translator"** provides a technical description of a prototypical translator named *bosto*—the BPEL to BPELscript Translator
- **Chapter 5 "Summary and Outlook"** concludes the work

Finally, the appendix shows language basics of BPELscript which are not mentioned in the text and the complete BPELscript grammar. Thereby, the grammar presents an overview of the correlation between SimPEL and BPELscript. Furthermore, it provides several overviews about expressions, annotations and keywords. For completeness and as it seems to be an emerging standard way when talking on BPEL, the appendix shows also a complete *Loan Approval Process* in BPELscript.

Chapter 2

Introduction to Computer Language Translation and the Main Tools

This work is about building a translator using automated tools. Therefore it is essential to understand formal language specifications and recognition to discover the "implicit tree structure behind input sentences and the generation of structured text." [Par07, p. 8]

Since professional and amateurish applications differ in the quality and correctness of the code, maintainability, reliability and in the strategy of error messages and recovery, it is obvious to use automated tools. Particularly with regard to maintainability it turns out that there is a family of tools that blend well with each other. These are ANTLR [Par07], ANTXR (ANother Tool for Xml Recognition [SE]), StringTemplate [Par06] and gUnit (Grammar Unit Testing [gUn]), mostly initiated by professor Terence Parr. Furthermore ANTLR-generated parsers automatically try to recover from syntax errors by performing single-symbol insertion or deletion upon mismatched symbol errors until they can restart in a known state [Par07, p. 231].

This chapter gives a short introduction to language translation basics and describes the above-named tools.

2.1 Basics of Language Translation

Mapping an input sentence of a language to an output sentence needs a translator which executes code that operates on the input symbols and emits output. For different sentences the translator must perform different actions, which means it must be able to recognize various sentences.

In the following this section explains only the basics of recognizing and translating text. For more detail on *compiler techniques* and for *translation* refer to [SLAU06], [WM96] or [Muc97].

2.1.1 Recognizing Language Syntax

Recognizing sentences is much like the human brain reads texts, not character by character but as a stream of words which are looked up in a "dictionary" before recognizing the grammatical structure [Par07, p. 4]. Therefore the recognition degenerates into two distinct phases, the lexical analysis and the syntactical analysis. The first is done by the lexer and the second by the parser. The *lexer* recognizes the structure of the incoming character stream and breaks it up into vocabulary symbols called tokens. The parser operates on the token stream and tries to recognize the grammatical sentence structure. Figure 2.1 illustrates the basic translation flow.

The formal language specification which describes the sentence structure is a set of rules called a grammar. This grammar could be recognized by a machine. For the generation of complex languages it turns out that machines without a memory are a too simple approach. Since sentences have an implicit tree structure behind [Par07, p. 21 f.] these machines cannot notice what they have generated in the past and thus generate invalid sentences. An example for this category are *state machines* which recognize *regular languages*. As memory system a conventional stack could be used which turns the state machine into a *pushdown machine* which generates only syntactically valid sentences.

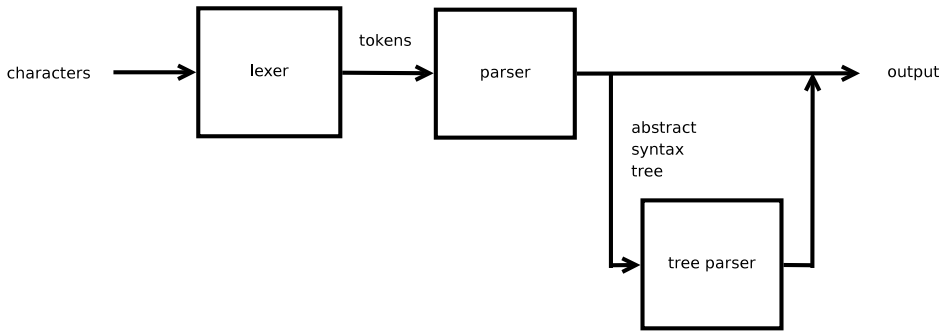

Figure 2.1: Overall Translation Process [Par07, p. 6]

There are two ways of recognizing text, top-down and bottom-up. The top-down recognition process starts at the most abstract language level. A common example for this class is called *recursive-descent* recognizer. The idea of recursive-descent parsing is to transform each rule of a grammar into a method that will recognize exactly that rule in the input. Since the input is recognized from **l**eft to right using the **l**eftmost derivation, the recognizer is called *LL*. Top-down recognizers walk the sentence tree structure in a depth-first manner. [Par07, p. 34] In contrast, bottom-up recognizers try to match the leaves at the bottom of the parse tree and work their way up to the starting rule. Since the input is recognized from **l**eft to right using the **r**ightmost derivation, the recognizer is called *LR*. Bottom-up recognizers consume input symbols until they find a matching complete alternative. [Par07, p. 35]

2.1.2 Translating Language Syntax

Back to translation it is usual to embed actions (e. g. code) which are executed according to the position within the grammar. For more complicated languages the translation is broken down into several

phases. Since it is absurd to reparse input character for each phase, it is more convenient to build an intermediate form and pass it between the phases. This intermediate form is a data structure called abstract syntax tree (AST) and is a highly processed and condensed version of the input. [Par07]

Depending on the complexity of the translation there are two main classes of translators. For simple languages a translator can execute actions that immediately emit output. For more complicated languages translators use the parser only to construct ASTs which are then scrambled by *tree parsers* to extract information needed by future phases. [Par07, p. 6] This approach requires a final *emitter* phase which generates structured text output to complete the relationship between characters, tokens and ASTs.

This section gave an overview about basics of language translation. The next section introduces ANTLR, a tool which can automatically generate lexer, parser and tree parser from a grammar.

2.2 ANTLR—ANother Tool for Language Recognition

This section introduces ANTLR—ANother Tool for Language Recognition [Par07]. ANTLR is a LL(*) [1] recursive-descent parser generator written by Terence Parr, professor of computer science at the University of San Francisco.

Since grammars are the formal specification of languages, ANTLR provides a framework to generate recognizers, compilers and translators from them. Thus it automates the construction of language recognizers containing actions, support for intermediate-form tree construction, tree walking and automatic error recovery. In contrast to other parser generators such as YACC (Yet Another Compiler Compiler [JJ75]), ANTLR generates top-down parsers which transform each rule of a grammar into a method and so is human readable. Furthermore it provides a combined DSL for both lexer and parser which is designed to make recognizers and translators easy to build. Thereby the recognizer will also try to recover and resynchronize by skipping tokens until it sees a token in the proper *following set*. [Par07, p. 49 ff.]

The easiest way to work with ANTLR grammars is to use ANTLRWorks [Par07, p. 46], which is a complete GUI development environment written by Jean Bovet, an ex-master student of Terence Parr. ANTLRWorks [PB07] provides a grammar editor with refactoring and navigation features, a grammar interpreter, and a grammar debugger.

In the following, the use of ANTLR grammars, attributes and actions, tree construction and tree grammars as well as generating structured text with templates is explained. The overall parsing process of ANTLR to translate BPELscript to BPEL is shown in Figure 2.2. Since the translation from BPEL to BPELscript deals with XML it is handled in a different way using ANTXR. This is explained in Section 2.3.

[1] LL(*) allows arbitrarily lookahead and is one of ANTLR's key features.

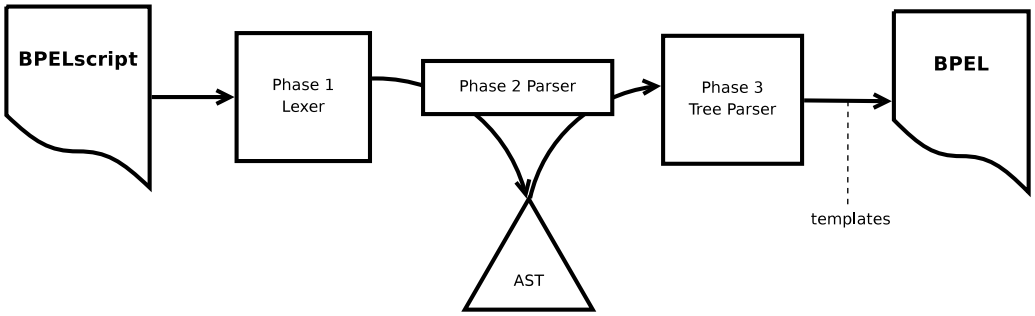

Figure 2.2: ANTLR Parsing Process

2.2.1 ANTLR Grammars

Section 2.1.1 introduced grammars as a list of rules describing the structure of a language. To specify the lexical and grammatical structure ANTLR uses a combined grammar for both parser and lexer rules. From this combined grammar, ANTLR generates the lexer and the parser. [Par07, p. 45 f.]

In ANTLR lexer rules begin with an uppercase letter and typically refer to character and string literals. In contrast to lexer rules, parser rules begin with a lowercase letter and typically refer to other parser rules, lexer rules or strings. Listing 2.1 shows an example of parser and lexer rules out of the BPELscript grammar. Each rule is defined by a rule name and a description separated by a colon. Thereby single quotes mark strings. Optional occurrences are marked with a question mark operator. Multiple, non empty occurrences are marked by the plus operator and multiple occurrences by the star operator. Alternatives are indicated by the pipe operator. Each rule ends with a semicolon.

Listing 2.1 Example ANTLR—Lexer and Parser Rules

```
// parser rule
ns_id : (pr=ID '::')? loc=ID

// lexer rule
ID : (LETTER | '_' ) (LETTER | DIGIT | '_' | '-' )*;
```

As top-down parser ANTLR generates a method for every rule defined in the grammar. Rule references are translated to method calls, and token references are translated to *match(TOKEN)* calls [Par07, p. 48 f.].

2.2.2 Attributes and Actions

Since a parser checks only syntactical correctness ANTLR does not emit any output without actions. To extract information or move from a recognizer to a translator or interpreter, code has to be embedded in the grammar. Within the generation ANTLR inserts the actions right after it generates code for the

preceding element [Par07, p. 55]. Thus, grammars are used to match something and embedded actions [Par08] are used to evaluate something.

A namespace identifier in BPELscript consist of an optional *prefix* followed by a double colon and a *location*. This is shown in Listing 2.1. The corresponding rule from the tree parser is shown in Listing 2.2. There are two interesting points in the example. The first is the usage of named attributes in case of multiple occurrences of the same token such as the *ID* token. Therefore each token is prefixed by an unique identifier which can be used by $*pre.text* for example. The second is the usage of actions which are encapsulated in curly braces. In the example the action part sets the return values of the rule. The enhanced method of naming attributes with the list operator is shown in Listing 2.3. The meaning of the elevated dash to specify AST nodes and the arrow operator to call a template are discussed in the following sections.

Listing 2.2 Example ANTLR—Attributes and Actions

```
ns_id returns [String nspre, String nsloc]
    : ^(NS pre=ID? loc=ID)
    {
      if ($pre != null) {$nspre = $pre.text;}
      $nsloc = $loc.text;
    }
    -> ns_id(pre_opt={$pre},loc={$loc.text});
```

2.2.3 Tree Construction

Section 2.1.2 discussed that for complicated languages translators use parsers only to construct an AST, separating parsing and evaluation into two phases. Therefore ANTLR provides a declarative tree construction mechanism which is embedded in the grammar and is much smaller and faster to read and write than embedded actions [Par07, p. 60].

The tree construction operator is the rewrite rule indicated by the arrow operator which can be also viewed as grammar-to-grammar transformation. Thus it is possible to extend a parser grammar so that it will construct an AST. To specify a tree structure simply indicate which tokens should be considered operators (subtree roots)—shown by elevated wedge in Listing 2.3*a*—and which tokens should be excluded from the tree—shown by exclamation point in Listing 2.3*b*. [Par07, p. 60 f.].

A program in BPELscript consists of at least one *declaration*. Therefore Listing 2.3*a* shows the corresponding parser rule. After the description of the rule appearance, the arrow operator introduces the rewrite rule to construct AST nodes for all declarations. Thereby, all program nodes are tagged by *ROOT* and contains multiple declarations. In contrast to the inclusion of tokens, Listing 2.3*b* describes the exclusion of tokens. Therefore the *variables* rule is taken. The definition of variables in BPELscript is introduced by the *var* tag which is followed by a list of variable declarations. Since it is unnecessary to keep the *variables* list in the AST only the *variables* itself are included in its corresponding rule. Furthermore, the example illustrates, that not all rules must have a rewrite rule. The next subsection explains why the *variables* rule of the example will disappear in the tree grammar.

Listing 2.3 Example ANTLR—Inclusion and Exclusion of Subtrees

```
// a - using '^' for subtree with ROOT as root and declarations as children
program : declaration+ -> ^(ROOT declaration+);

// b - using '!' to avoid building nodes
variables : 'var'! v+=variable (','! v+=variable)*;
```

2.2.4 Tree Grammars

From a tree grammar, ANTLR can generate a tree walker using the same top-down recursive-descent parsing strategy used for the lexer and parser [Par07, p. 60].

To walk the AST of the previous subsection, a tree grammar can be derived from the parser grammar by removing the parser fragments left of the rewrite operator. Also the lexer rules have to be removed because a tree grammar operates on tokens not on characters. ANTLR will then generate a tree parser from the grammar automatically. The example from 2.2.3 degenerates to only one rule because the *variables* rule has no rewrite rule—this is shown in Listing 2.4. Obviously, the notation of a tree grammar is identical to the regular ones, except from the introduction of a two-dimensional tree construct. [Par07, p. 58 ff.]

Listing 2.4 Example ANTLR—Deriving a Tree Grammar from Listing 2.3

```
program : ^(ROOT declaration+);
```

Since Section 2.1.2 states that the complex approach requires a final emitter phase to generate structured text output, the following section explains ANTLRs possibilities to generate structured text output with templates.

2.2.5 Generating Structured Text with Templates

To reduce complexity of the translation it is a common approach to process and condense the input into an AST. However, "no translator can escape the final phase: generating structured text from [...] internal data structures." [Par07, p. 195]

Restricted to embedded actions translators must emit code with print statements using arbitrary code to generate output. Since this is very painful and error prone [Par07, p. 198] it is more convenient to use preformulated templates with holes to stick in some values from the AST. Hence it seems to be obvious to embed template construction actions in a grammar just like other actions. For this the rewrite operator -> was overloaded and a separate library called StringTemplate was added to ANTLR which is explained in Section 2.4. To renew the example of the previous sections, the *program* rule in the tree grammar is extended with a template call on the template *list* to process some output within the template.

Once again, the notation for a tree grammar is identical to the regular ones but with a different meaning. Since this seems to be confusing at the beginning, the following example points out the relation. In

Listing 2.5 Example ANTLR—Extending the Tree Grammar from Listing 2.4 with templates

```
program
     : ^(ROOT decls+=declaration+)
     -> list(content_st={$decls});
```

Listing 2.6 there are two versions of the same rule. The first one is a parser rule which resorts the attributes to build an AST subtree with *PROC_STMTS* as root and optional joins, multiple signals and a procedure statement. The second one is a tree parser rule where the template *list* is called with the template of the procedure statement.

Listing 2.6 Example ANTLR—Template Construction

```
// parser rule with rewrite rule
proc_stmts
     : (join SEMI)? proc_stmt (s+=signal SEMI)*
     -> ^(PROC_STMTS join? signal* proc_stmt);

// tree walker rule with template construction
proc_stmts
     : ^(PROC_STMTS j+=join? s+=signal* proc_stmt[$j,$s])
     -> list(content_st={$proc_stmt.st});
```

This section demonstrated that using ANTLR grammars with rewrites and templates is a great way for generating translators.

The handling of XML with XPA (XML Processing for ANTLR) [XPAa] requires verbose specification and has no support for namespaces [SE]. Thus the next section introduces ANTXR which is a fork from ANTLR.

2.3 ANTXR—ANother Tool for Xml Recognition

ANTXR [SE] is a Parser Generator for XML recognition written by Scott Stanchfield. It is based on the ANTLR Parser Generator. The main idea is to extend conventional XML parsing mechanisms with ANTLR-like context handling. Thus, XML parsing rules can solve many problems as an ANTXR grammar by producing an effective and maintainable XML parser. [SE] In the future it is planed to have an easy updateable parser which is in sync with the XML schema.

ANTXR is used for translating BPELscript. There are three main reasons to use ANTXR:

- Same techniques
- Understandable grammar syntax
- XSD Support in future

Using the same technique for translating BPEL to BPELscript as for the translation of BPELscript to BPEL is fundamental for maintainability. Furthermore, it is important to use an understandable grammar

syntax for handling XML. Therefore, an easy handling of start and end tags and an easy attribute access is essential. There are two possibilities of XML handling with ANTLR-like techniques.

- XPA—XML Processing for ANTLR [XPAa]
- ANTXR—ANother Tool for Xml Recognition

First, ANTLR itself provides a possibility for XML processing named XPA. The handling of XML with XPA requires verbose specification (e. g. explicit tag handling and complicated attribute access) and has no support for namespaces. Second, ANTXR provides an easy understandable grammar syntax (e. g. implicit tag handling and simple attribute access) with namespace support. [SE] Thus, this work uses ANTXR for the translation.

XML parsers must be in synch with its XSD schema. Therefore, it is important to represent the parser in a form parallel to the schema. This exculpate users from doing this manually which is inefficient and error prone. A main goal of future development of ANTXR is the capability to generate an ANTXR grammar from an XML Schema.

A disadvantage is, that ANTXR is still based on ANTLR v2.7. Hence, it does not provide the the usage of templates, so that users have to embed their actions directly in the grammar. Unfortunately, this restricts the retargetability and is error prone.

This section gives a short introduction to ANTXR. It explains the intermediate representation of XML, ANTXR grammars and how to generate structured text with embedded actions.

2.3.1 Intermediate Representation of XML

The first thing on intermediate representation of XML is the question of lexing XML. The main possibilities are DOM (Document Object Model) [DOM] and SAX (Simple API for XML) [SAX] which are explained in the following.

A DOM parser reads XML into memory and converts it into a tree of DOM nodes. The larger the XML document, the more memory this takes. A SAX parser is an event-based mechanism that pushes the data when it sees tags and content. It is pretty memory-efficient but extra context information has to be tracked by the user. When lexing XML it is a basic necessity to provide a parser which is fast, robust and has optional validation. ANTXR assumes that SAX parser are the best solution for this, because "they're being pounded on by thousands of developers". The main goals of ANTXR are to provide a declarative specification of how to process XML, define the handling of certain patterns including context and to keep track of the context appropriately. This was achieved by forking from ANTLR to gain a clear XML-like syntax. [SE]

The XML parsing process with ANTXR is shown in Figure 2.3. For a proper handling, BPEL's *ExtensionActivity* requires a *preprocessing* to reorder XML elements. This optional *preprocessing* is explained later in Section 4.2. The lexing part is done by a *SAX front-end* which can be chosen and configured by the user. After that comes the *XMLTokenStream*, which is a handler for the SAX parser and a source for the ANTLR parser. Since SAX is a "push" API and ANTLR is a "pull" API, the *XMLTokenStream* merges these concepts by supplying a blocking queue. [SE] The ANTLR Parser retrieves tokens from the blocking queue and translates them into BPELscript with embedded actions.

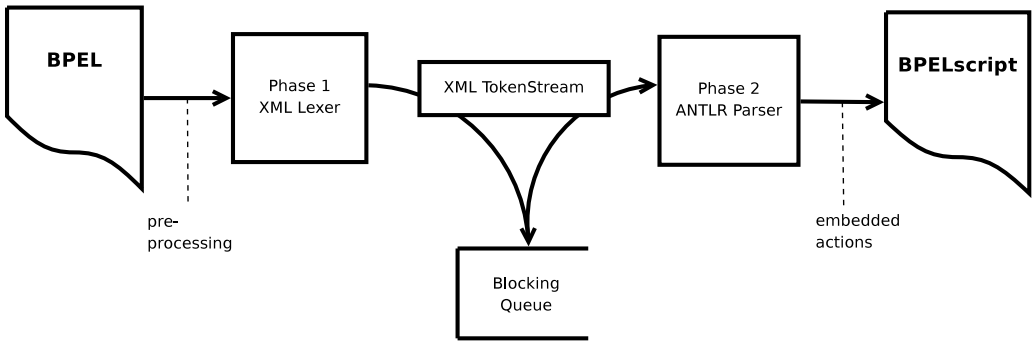

Figure 2.3: ANTXR Parsing Process

2.3.2 ANTXR Grammar

The main constructs are the same since ANTXR is a fork of ANTLR. Thus, this section provides a short summary and an example of ANTXR.

The star operator identifies zero or more of the enclosed symbols, the plus operator one or more of the enclosed symbols and the question mark tags the enclosed symbol as optional. $x \mid y$ designates either x or y appearance and action code that gets executed within a rule is enclosed by curly brackets. References to XML attributes are done with the text circled *a* and references to XML tags are done with angle brackets such as <*name*>. A symbol is a reference to a grammar rule which is not an XML tag (i. e. it has no angle brackets). Finally *PCDATA* stands for a reference to character data.

To get an impression of ANTXR grammars Listing 2.7 shows a simple XML chunk which can be parsed by the ANTXR grammar in Listing 2.8.

Listing 2.7 Example ANTXR—a simple XML chunk (based on [SE])

```
<?xml version="1.0"?>
<people>
  <person ssn="111-11-1111">
    <firstName>Frank</firstName>
    <lastName>Leymann</lastName>
  </person>
  <person ssn="222-22-2222">
    <firstName>Marc</firstName>
    <lastName>Bischof</lastName>
  </person>
</people>
```

Listing 2.8 Example ANTXR—a simple XML Parser for the chunk in Listing 2.7 [SE]

```
header {package com.javadude.antlr.sample.xml;}

class PeopleParser extends Parser;

document : <people> EOF;

<people> : (<person>)*;

<person> : ( <firstName> | <lastName>)*;

<firstName> : PCDATA;

<lastName> : PCDATA;
```

2.3.3 Generating Structured Text with Embedded Actions

Embedded actions in ANTXR are basically the same as in ANTLR. The difference is, that ANTXR does not allow the use of StringTemplates, since ANTXR uses ANTLR V2.7. To get an impression of embedded actions in ANTXR Listing 2.9 shows a chunk of the ANTXR BPEL grammar used in the BPEL to BPELscript translation. The example shows the rule of the join condition XML tag. The rule has the string *text* as return result, an attribute reference to *expressionLanguage* and contains some *PCDATA* which are converted in the action.

Listing 2.9 Example ANTXR—using Embedded Actions

```
<joinCondition> returns [String text=""]
  {
    String exprLg = @expressionLanguage;
  }
  : boolExpr:PCDATA
  {
    text+="["+boolExpr.getText().trim()+"]"; //convert PCDATA
  };
```

This section explained the main details to process XML with an ANTLR-like parser generator. The next section presents an important part of ANTLRs strategy to emit formatted text output with an declarative template language: StringTemplate.

2.4 String Template Engine

StringTemplate [Par06] is a java template engine for generating formatted text output which is particularly good at multi-targeted code generators by strictly enforcing the model-view separation. A template engine is a simple code generator that emits text using templates, which are just documents with placeholders where one can stick values. [Str] StringTemplate is included in ANTLR since version

3.0 and can be used via the embedded action mechanism like this is done in BPELscript for the code generation. Section 2.2.5 described how to use templates with ANTLR.

This section gives a short introduction to StringTemplate and explains the four template constructs which Terrence Parr thinks are the only needed ones [Par04, p. 2]. These are attribute reference, conditional template inclusion, nested template references and template application to a multi-valued attribute similar to lambda functions [2] and LISP's [3] map operator [Par04].

2.4.1 Defining Templates

A StringTemplate breaks up the templates into two chunks. These are *text* and *attribute expressions*. Attribute expressions are enclosed by angle brackets such as <*attribute-expression*>. When evaluating a template, *text* is interpreted as text and *attribute expressions* are replaced by its value (e. g. derived to a string). The templates are stored in a StringTemplate group file with the file extension *.stg*.

For a first example, suppose a template to compose a letter to some friends. Therefore the template in Listing 2.10 contains two strings ("Dear" and "...") and an attribute expression (<*friends*>), which holds a list of names. For simplicity, suppose that the list is evaluated to a comma separated list of names (this is explained in detail in Section 2.4.1). After the evaluation, the attribute expression is replaced by its value like shown in the same figure.

Listing 2.10 Example StringTemplate—Easy Template Definition

```
# template
  Dear <friends> ...

# result
  Dear Olly, Tammo, Marc ...
```

There are three types of templates: anonymous, single-line and multi-line template definitions which are shown in Listing 2.11. Single-line templates are useful for short descriptions not containing any quote characters. In case of quote characters, multi-line templates are the preferred solution. Line wraps should be treated with caution, because StringTemplate supports automatic line indentation. The line indentation takes each whitespace after a newline and prefixes the following characters with these whitespaces. This may be hard to understand when using recursive subtemplates which uses such auto indentation. In addition, there is a rule which allows to structure the templates: kill a single newline after <*if*>, *«*, <*else*>, and <*endif*> (but for <*endif*> only if it is on a line by itself) [ANT].

Attribute Reference

The use of attributes allows two main types of references: *attribute reference* and *property reference*. An *attribute reference* is a reference to the (string) value of an attribute such as the *friends list* denoted earlier. A *property reference* is a reference to a part or property of a reference. Therefor, suppose a

[2] For details refer to the lambda calculus of mathematical logic and computer science [BBB+97].

[3] A multi-paradigm programming language with dynamic strong typing, mainly developed by Steve Russell [McC60].

Listing 2.11 Example StringTemplate—Template Definitions

```
# anonymous template
{ ... }

# single-line template
template(argument-list) ::= " ... "

# multi-line template
template(argument-list) ::= <<...>>
```

person structure or record containing a *first* and a *last* name. With a property reference it is possible to specify a reference to the first name by <*person.first*>.

<*X*> is evaluated to *X.toString()*. If the attribute is not given in the XML file, <X> evaluates to the empty string. Analogous <*X.property*> evaluates to the value of *X.property.toString()*. If the attribute is not given or the property is not found in the XML file, <X.property> evaluates to the empty string.

Conditional Behavior

Conditional behavior is based upon the presence of an attribute and allows the common use of if statements. An example is shown in Listing 2.12 and has the following meaning. If the attribute *x* has a value or is a boolean object that evaluates to true, *subtemplate* is included. Else, if *y* has no value or is a boolean object that evaluates to false, *subtemplate2* is included. If neither of them is executed, *subtemplate3* is included.

Listing 2.12 Example StringTemplate—Conditional Behavior

```
<if(x)>subtemplate
<elseif(!y)>subtemplate2
<else>subtemplate3
<endif>
```

Nested Template References

StringTemplate uses dynamic scoping for templates, since this is natural for template engines. Thus, a nested template may see any attributes of the enclosing instance or its enclosed instances. In contrast to lexical scoping used in most programming languages, a method may not see the variables defined in invoking methods. [ANT]

Referencing to an attribute *attr* in a template is resolved in four steps. First, it is looked up in the templates attribute table and its arguments. After that it is looked up recursively in the templates enclosing template instance chain and then up in the templates group or supergroup chain for a map. As a result one must beware of infinite recursion in the case of heavily nested template tree structures. To

prevent this, formal arguments on templates hide any attribute values with that name in any enclosing scope. [ANT] Figure 2.13 shows common use cases of nested template references.

Listing 2.13 Example StringTemplate—Template Reference

```
# a - template call with expression
<template(template_arg=expr)>

# b - template call with other template
<template(template_arg=otherTemplate)>

# c - template call with anonymous template
<template(template_arg={...})>
```

Template Application to Multi-Valued Attributes

Multi-Valued Attributes (MVA) are a very important and powerful template construct particularly with regard to list processing. They are similar to lambda functions. Common examples of MVAs are shown in Listing 2.14 and explained in the following.

Listing 2.14 *a* presents simple call of a MVA being similar an attribute. This results in a concatenation of the result of *toString()* invoked on each element. Thereby missing values evaluate to the empty string. Listing 2.14 *b* shows a call of a MVA which results in a listing where each element invocation is ended by a new line or any other expression. Listing 2.14 *c* shows the complex use of a MVA which could be a Java HashMap with keys and values. It is a parallel list iteration which walks through the values *keys* and *values* of the attributes, setting the values to the arguments in the anonymous template.

Listing 2.14 Example StringTemplate—Multi-Valued Attributes

```
# a - simple multi-valued attribute
<multi-valued-attribute>

# b - multi-valued attribute separated by new line
<multi-valued-attribute; separator="\n">

# c - applying an anonymous template to a multi-value attribute separated by new line
<mva1.keys, mva2.values:{k, v | do something with <k> and <v>}; separator="\n">
```

This section introduced StringTemplates template definition. The next section introduces another part of grammar development projects: grammar unit testing.

2.5 GUnit—Grammar Unit Testing

GUnit [gUn] is an unit testing framework for ANTLR grammars written by Leon Jen-Yuan Su. It provides a simple way for automating test cases for grammars in a manner similar to what *jUnit*

[Fow99] does for unit testing of *Java* programs [gUn]. The basic idea is to come up with a clean syntax such as *input → expected output* for rules in a grammar. In that way "it's just what unit testing is for: testing code [...] AND documenting it with examples" [Via08]. *bosto* uses gUnit to provide a set of or examples and to test the correctness of the translation.

This section explains how to use gUnit and how it was adapted to meet the projects needs.

2.5.1 JUnit-based Grammar Testing

The core concept behind gUnit is to provide a DSL for grammar unit testing with two main functions, interpreter and jUnit generator.

The interpreter interprets the gUnit test suite and runs the unit tests using Java reflection to invoke parser objects. The generator translates the gUnit test suite to jUnit Java code that can be compiled and executed by hand. [gUn] As a result, a test suite for a grammar with multiple unit tests per rule can be easy specified and translated into a jUnit test file. Thereby the input can be one out of single line, multiple lines or an external file. Additionally the output can be one of success, failure, an AST, a rule return value or some text output. The test checks whether the input causes the output. If ok (fail) is used, the test checks whether the input is (not) valid. An example for a gUnit test suite is shown in Listing 2.15.

Listing 2.15 Example gUnit Test suite [gUn]

```
gunit SimpleC;

variable: //test rule:variable with 2 tests
"int x" FAIL  //expect failure, because of missing ';' in the input string

"int x;" -> "x" //expect standard output "x" from rule

functionHeader: //test rule:functionHeader with 1 test
"void bar(int x)" returns ["int"] //expect a return string "int" from rule

program: //test rule:program with 3 tests, input starts immediately after the initial <<
          so the first test is a blank line
<<
char c;
int x;
>> OK       //expect success (no error messages from ANTLR)
```

2.5.2 Extending gUnit with Test of Templates

Template testing is a main requirement of text translation using StringTemplate. However, gUnit does not provide template testing yet. Therefore, a fork of gUnit has been realized. gUnit provides a good architecture, where the template processing is encapsulated in a single file. Thus, only the template file had to be changed. The gUnit template provides two main types of methods: *rule methods* and *parser methods*, each for parsers and tree parsers. Rule methods refer to success/fail tests and parser methods

refer to expected output tests. As a result, there are four main extensions which are explained in the following.

- Adding imports
- Load Template File
- readOutput Extension
- Pre-/Suffix Extension

First some imports are added which cannot be loaded by using the gUnit header option since they are ignored by the gUnit parser. They are shown in Listing 2.16 *a*. As next, it was necessary to load the template file from a given path which is shown in Listing 2.16 *b*. The test of outputs should be done preferably by using external files. Therefore, a *readOutput* method was added. Finally, pre- and suffixes are added to simplify the semantic of the input/output pairs. This allows to only specify a file name (without file extension) and a path where are two files, the filename.inputtype and the filename.outputtype. For example, a rule *reply -> "activities/"* results in a test, which checks whether the input *activities/reply.bpelscript* is translated into *activities/reply.bpel*.

Listing 2.16 Changes in the gUnit Template File

```
# a - adding imports
import iaas.bpelscript.antlr.BPELscriptLexer;
import iaas.bpelscript.antlr.BPELscriptParser;
import iaas.bpelscript.antlr.BPELscriptWalker;
import iaas.antlr.stringtemplate.StringTemplateGroup;

# b - adding template loading
FileReader groupFileBPELscript = new FileReader(templatePath);
StringTemplateGroup templates = new StringTemplateGroup(groupFileBPELscript);
groupFileBPELscript.close();
```

This introduction gave an overall view of language translation basics and the main tools which are used for the translation from BPEL to BPELscript and vice versa. The next chapter describes this translation in detail.

Chapter 3

BPELscript

BPEL is one of "the most prominent business process notations in use today" [KMWL08] and therefore qualified for the technical implementation of business processes and orchestration of services respectively. It aims to enable *programming in the large* using the *universal data speech* XML. Since XML is no programming language at all [Hid07], BPEL leaks rapid prototyping without visual builders. Therefore the Apache ODE Group [ODEb] proposed a lightweight scripting language named *SimPEL* (Simplified BPEL Syntax) [BAR, LK, RE]. The goal of SimPEL is to provide a workflow language similar to BPEL, but with a simplified syntax and simplified semantics. For example, SimPEL changes the correlation mechanism of BPEL [Rio08]. Therefore, SimPEL scripts cannot be translated to BPEL, which makes it impossible to run them on BPEL engines not supporting SimPEL.

To provide a simple syntax for BPEL on the one hand and to provide an integration to each standard-compliant BPEL engine, this work takes a different approach. This approach adopts SimPEL's main ideas without changing the semantics of BPEL. The resulting language is called *BPELscript*. BPELscript offers the same semantics as BPEL and is fully translatable to WS-BPEL 2.0. In addition, each WS-BPEL 2.0 process is fully translatable to BPELscript.

This chapter provides a bijective translation function from BPEL to BPELscript and vice versa. The following sections presents how all parts of BPEL processes are modeled in BPELscript, showing examples and covering modeling alternatives where necessary. Thereby the term *activity* refers to BPEL's activities, whereas the term *statement* refers to the equivalent in BPELscript. BPELscript introduces a simplified syntax for BPEL. Thus, it is a main requirement to use the same keywords to introduce a statement as in its corresponding BPEL activity. This term is referenced to, by the *preluding keyword approach*. A complete overview, indicating the correlation between SimPEL and BPELscript is presented in Appendix A.6 on Page 96.

3.1 The Structure of Processes in BPELscript

To get a first impression of how processes look in BPELscript, Listing 1.3 illustrates how the *Loan Approval Process* from [OAS07a, 15.3.2] looks in BPELscript. For simplicity some details are omitted here, but nevertheless, for this example, BPELscript is about three times smaller than the original BPEL code. For the complete example refer to appendix A.4 on page 95.

Listing 3.1 presents a conceptual BPEL process and Listing 3.2 its BPELscript equivalent. BPEL processes are enclosed by a *process* tag, containing a data part and an activity part. This tag provides attributes such as the mandatory process name, the target namespace and the XML namespace. Furthermore, the optional attributes *queryLanguage*, *expressionLanguage*, *suppressJoinFailure* and *exitOnStandardFault*. The data part provides a centralized definition part for *extensions*, *imports*, *partnerLinks*, *messagesExchanges*, *variables*, *correlationSets*, *faultHandlers* and *eventHandlers*. Finally, the activity part provides the process behavior.

Formally, a BPELscript process consists of three parts: header, middle and footer. The header contains *namespace*, *extension* and *import* declarations. The middle part contains the process itself and the footer contains the *event handling*. The following sections of this chapter will explain all parts in detail. Note, that this work only translates executable processes (For proposals to handle abstract processes refer to Section 4.3.1).

Listing 3.1 The Structure of a BPEL Process

```
# BPEL
  <process name="NCName" targetNamespace="anyURI"
     queryLanguage="anyURI"? expressionLanguage="anyURI"?
     suppressJoinFailure="yes|no"? exitOnStandardFault="yes|no"?
     xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable">

   <extensions/>?
   <import/>*
   <partnerLinks/>?
   <messageExchanges/>?
   <variables/>?
   <correlationSets/>?
   <faultHandlers/>?
   <eventHandlers/>?
   activity
  </process>
```

Listing 3.2 The Structure of a BPELscript Process

```
# BPELscript
  # header
  (namespace | extension | imports)?
  # middle
  process tns::Name {
     activity
  }
  # footer
  eventHandler
```

3.2 Relationship to Business Partners—Partner Links

With BPEL it is possible to model the relationships between partner processes by aggregating web services and orchestrate their service interactions. These connection between a process and its partners is described with *partnerLinks* which can be regarded as communication channels between business partners. The functionality of each partner link is defined by its *partnerLinkType* and its *role name*. [OAS07a, OAS07b, 6, 3.2] The BPEL definition of partner links and its types is shown in Listing 3.3. Note, that partner link types are declared in the WSDL document and partner links are instances of partner links and declared in the BPEL process. For the handling of BPEL *partnerLinks* in BPELscript, there are three solutions which are discussed in the following.

Listing 3.3 BPEL Partner Links and Partner Link Types

```
# WSDL Definition
<partnerLinkType name="NCName">
   <role name="NCName" portType="QName"/>
   <role name="NCName" portType="QName"/>?
</partnerLinkType>

# BPEL Instance
<partnerLinks>
   <partnerLink name="NCName" partnerLinkType="QName"
      myRole="NCName"? partnerRole="NCName"?
      initializePartnerRole="yes|no"? >+
</partnerLinks>
```

The first idea of handling partner links in BPELscript is to provide a partner link definition with the preluding *partnerLink* keyword and a list of partner links following. Additionally content is stored direct after the particular partner link name in brackets. Thereby annotations, like they are introduced later, does not provide the expected semantic. In this way, BPELscript supports the *initializePartnerRole* attribute inside the bracketing block. The outcome of this are two solutions. Solution one is shown in Listing 3.4 *a* and is similar to the BPEL specification. It shows a mandatory partner link type and three optional parameters which specify the *myRole*, *partnerRole* and the *initializePartnerRole* attribute (Annotations are discussed in detail in Section 3.4.1). The problem is, that one will need an extra command in BPELscript to specify the partner link types if they are not imported. Therefore, this solution is not used, but parts of it are reused in solution two.

Solution two is shown in Listing 3.4 *b* and comes with two variants. The idea of the first variant is to omit the partner link type indirection. This is done by omitting the partner link type at the beginning and specify the port types directly. When using existing partner link types, this requires the user to specify the port type multiple times. Thus, the idea of the second variant is to reuse the idea of the first solution without any separate partner link type declarations.

The third solution is similar to solution *b.1*. It splits the client role (*client* keyword) and the partner role (*service* keyword) into two chunks like this is shown in Listing 3.4 *c* [LK]. Such as solution *b.1*, this approach misses the usage of existing partner link types since it uses port types directly. Thus, it will require an additional partner link type *import statement*. As a result, this will introduce three non-BPEL conform statements.

Thus, BPELscript uses the second variant of solution two as mixture of solution *a* and *b* in Listing 3.4 *b.2*. Thereby it provides usage of existing WSDL port types and a short notation without the indirection (i. e. when using an existing WSDL port type, one has only to specify the roles if wanted or null else. When using the short notation, the optional first part (the partner link type) is not used, but the port type specification.). In a process, partner links are defined like variables everywhere in the process. Similar to variables (presented in Section 3.3.1), the translation collects them and move them to the globals part of its enclosing scope or process. In Section 3.3.1 this work discuss an implicit variable declaration. For partner links, an implicit declaration has to provide the same attributes as the explicit definition. Additionally, it is difficult to derive the necessary information from the left side of the assignment. Thus, this is not supported since in BPELscript.

Listing 3.4 Handling of Partner Links

```
# solution a
  partnerLink plink = (ns::plType, roleA?, roleB?, @initializePartner?);

# solution b.1
  partnerLink plink = (ns::portType roleA, ns::portType roleB);

# solution b.2
  partnerLink plink = (ns::plType?, ns::portType?, ns::portType?, @initializePartner?);

# solution c
  //process portType
  client roleA = ns::portType
  //partner portType
  service roleB = ns::portType;
```

3.3 State of a BPEL Process—Data Handling

The previous section introduced partner links as possibility to declare relationships to external partners. This section will discuss the handling of *process data*, which are variables holding the data that constitute the state of a process [OAS07b, 3.].

3.3.1 Variables

Variables must have a type (WSDL message type, XML simple/complex type or schema elements). The syntax of a variable declaration in BPEL is shown in Listing 3.5 also an example of an explicit variable declaration in BPELscript is given.

BPELscript adopts the implicit variable declaration from SimPEL. Implicit variable declaration means, that variables are declared by an assignment for example. Assignments in BPELscript are discussed in detail in the next section. Similar to partner links, variables can be defined explicitly everywhere in a process or scope. A translation collects them and moves them to the respective part of its enclosing scope or process. Since each variable belongs to a scope, each variable is either global to the process

Listing 3.5 Handling of Variables

```
# BPEL
  <variables>
     <variable name="BPELVariableName"
        messageType="QName"? type="QName"?
        element="QName"?>+
        from-spec?
     </variable>
  </variables>

# BPELscript
  var @msgType "QName" @type "QName" @element "QName" var1,
     var2;
```

or local to its enclosing scope. To simplify matters, variables derived from implicit assignments are untyped yet and have non of its BPEL attributes. [1]

3.3.2 Assignment

The BPEL specification states, that "[the *assign*] activity can be used to copy data from one variable to another, as well as to construct and insert new data using expressions." [OAS07a, 8.4] Therefore it copies a value from the source (*from-spec*) to the destination (*to-spec*). Since the BPEL specification does not require validation of variables at the time of writing, it cannot be guaranteed that they are valid according to their definitions. Thus, BPEL provides the *validate attribute* for validation of variables against their associated XML and WSDL data definitions (see also the *validate* activity in Section 3.4.7). Since BPELscript is block-structured it provides assignments as usual with $lvalue = rvalue$ [2]. Listing 3.6 shows the BPEL specification of assignments and how it is realized in BPELscript.

Variations of Assignments

BPELs *From* and *To* specification are shown in Listing 3.7 and discussed in the following.

The simplest form of an assignment is to overwrite the value of a variable with a new literal. In BPELscript this is realized as $lvalue = 'literal'$. BPELscript adopts arithmetic and logical expressions from SimPEL. [3] When a literal is such an expression it is currently not handled separately (i. e. right hand sides of assignments are treated as literal such as shown in Listing 3.8).

If variables are affected by a path reference, it must be distinguished between a *part* and a *property* reference. Per default, path references in BPELscript are references to parts. A property reference must be followed by an *@property* annotation. This annotation can be omitted if BPELscript has an own type system in the backend which can infer this information on its own or if the E4X-extension

[1] To increase usability this has to be done in future work by adding a type system to BPELscript which infers the type of variables. Cf. Chapter 5 on Page 84—Variables.

[2] lvalue/rvalue denotes the left (right) hand side value of the assignment

[3] For an overview about expressions, refer to Appendix A.2 on Page 92.

Listing 3.6 Translation of Assign

```
# BPEL
  <assign validate="yes|no"? standard-attributes>
     standard-elements
     (<copy keepSrcElementName="yes|no"? ignoreMissingFromData="yes|no"?>
        from-spec
        to-spec
      </copy>
     |<extensionAssignOperation>
        assign-element-of-other-namespace
      </extensionAssignOperation>
     )+
  </assign>

# BPELscript
  to-spec = from-spec;
```

Listing 3.7 Definition of From and To Specification

```
# a - from spec
  <from><literal>literal value</literal></from>
  <from variable="BPELVariableName" part="NCName"?>
     <query queryLanguage="anyURI"?>?
        queryContent
     </query>
  </from>
  <from variable="BPELVariableName" property="QName" />
  <from partnerLink="NCName" endpointReference="myRole|partnerRole" />
  <from expressionLanguage="anyURI"?>expression</from>

# b - to spec
  <to variable="BPELVariableName" part="NCName"?>
     <query queryLanguage="anyURI"?>?
        queryContent
     </query>
  </to>
  <to variable="BPELVariableName" property="QName" />
  <to partnerLink="NCName" />
  <to expressionLanguage="anyURI"?>expression</to>
```

(ECMAScript for XML [ECM]) is used. An example of path reference assignments and its BPEL equivalents is shown in Listing 3.9.

Listing 3.10 gives a more complex example of path reference assignments (assume that *v1* is defined anywhere). Furthermore, a proposal of a direct translation in XPath [XPab] references is shown.

Listing 3.8 Handling of Arithmetic/Boolean Expressions

```
# a - BPELscript
  v2 = 5*v1;

# b - equivalent BPEL
  <assign>
    <copy>
      <from>
        <literal>
          5*v1
        </literal>
      </from>
      <to variable="v2"/>
    </copy>
  </assign>
```

Listing 3.9 Path Reference Assignments in BPELscript and its BPEL Results

```
# a - simple variable assignment
  var1 = var2;

# b - using default part reference and property annotation
  var1 = var2.attr1;
  var1 = var2.part1 @property;

# BPEL equivalents
  <assign>
     <copy>
        <from variable="var2"/> // property="attr1" or part="part1"
        <to variable="var1"/>
     </copy>
  </assign>
```

Listing 3.10 Complex Path Reference Assignments in BPELscript and its BPEL Results

```
# a - complex variable assignment using parts
  v2.h.i.j = v1.b.c.d.e;

# BPEL equivalents
  <assign>
     <copy>
        <from variable="v1" part="b.c.d.e"/>
        <to variable="v2" part="h.i.j"/>
     </copy>
  </assign>

# or using XPath (currently not supported)
  <from>$v1/b/c/d/e<from>
  <to>$v2/h/i/j<to>
```

Assignments and PartnerLinks

Assignments which affect partner links are similar to variables since *endPointReferences* have the same syntax as parts and properties. It is important to specify the end point reference because it is a mandatory attribute. If a partner link is declared in the process, but the *endPointReference* is missing, the rvalue of the assignment is interpreted as literal like in Listing 3.11 *a* and the assignment itself will declare a implicit variable named by the lvalue. BPEL constraints the *toPart* of partner link assignments to the *partnerRole*. BPELscript follows this constraint, but allows an additional specification of the partner role at the lvalue for readability. Listing 3.11 shows the different kinds of partner link assignments in BPELscript.

Listing 3.11 Partner Link Assignments in BPELscript and its BPEL Results

```
# a - missing endPointReference (role)
  plink1 = plink2;

# b - using default endPointReference of plink1
  plink1 = plink2.myRole;

# BPEL equivalents
  <assign>
     <copy>
        <from>
          <literal>
            plink2
          </literal>
        </from>
        <to variable="plink1"/>
     </copy>
  </assign>

  <assign>
     <copy>
        <from partnerLink="plink2" endpointReference="myRole"/>
        <to partnerLink="plink1"/>
     </copy>
  </assign>
```

Assignments and other Expression Languages

Listing 3.6 shows two ways of assignments:

- the copy activity
- the extensionAssignOperation

The *copy activity* provides a possibility to use an *expressionLanguage* with the *from* and *to* specification. BPELscript restricts itself to support only the *from* specification, since the usage of other expression languages in the *to* specification can be simulated by using the *extensionAssignOperation* discussed later in this section. Thus, BPELscript uses the *extensionExpressions* on a right side of an assignment,

which is enclosed by squared brackets [BAR]. The extension expression allows different kinds of expression languages which are translated directly.[4] This form of the assignment realized the <from experessionLangauge> part of BPEL's assignment. Listing 3.12 shows an example which uses *bpel:doXslTransform() XPath 1.0* extension function for transformation. Further assignments with the *receive* and *invoke* activity are discussed in Section 3.4.5.

Listing 3.12 Extension Expression Assignments and its BPEL Results (Cf. [OAS07a, 8.4])

```
# using extensionExpression in assignments
  B =  [bpel:doXslTransform("urn:stylesheets:A2B.xsl", $A)];

# equivalent BPEL
  <assign>
     <copy>
        <from>
          bpel:doXslTransform("urn:stylesheets:A2B.xsl", $A)
        </from>
        <to variable="B"/>
     </copy>
  </assign>
```

The *copy activity* supports an indirect use of *E4X*, such as shown in Listing 3.12. Additionally, the *extensionAssignOperation* is supported by using the *extensionActivity*, discussed in Section 3.4.11. Thereby, one can use both E4X and arbitrary expression languages such as shown in Listing 3.13.

Listing 3.13 Extension Expression Assignments supporting E4X

```
# using extensionAssignOperation with ExtensionActivity Statement
  {{{
     <extensionAssignOperation>
       <js:snippet>
         **Placeholder for the assignment**
       </js:snippet>
     </extensionAssignOperation>
  }}}

# equivalent BPEL
  <assign>
     <extensionAssignOperation>
       <js:snippet>
         **Placeholder for the assignment**
       </js:snippet>
     </extensionAssignOperation>
  </assign>
```

The advantage of extension expressions or assignment is its lower interpretation effort, but it requires support by the workflow engine according to the snippet.

[4] Translations should parse these chunks. With ANTLR this can be done with embedded *Island Grammars*, but this is not supported in *bosto* yet.

3.4 Basic Activities

To perform the process logic, BPEL provides two classes of activities: basic and structured activities. The elemental steps of a process are described with basic activities which are explained in this section. Structured activities describe the control flow and contain other activities recursively. They are described in Section 3.5.

3.4.1 Standard Attributes for All Activities

BPEL provides two optional standard attributes for each activity: *name* and *suppressJoinFailure*, which are shown in Listing 3.14 *a*. Thereby the name "is used to provide machine-processable names for activities" [OAS07a, 10.1] and the *suppressJoinFailure* attribute indicates "whether a join fault should be suppressed if it occurs" [OAS07a, 10.1].

Standard attributes are not supported by SimPEL. Since it is too restrictive to provide a simplified BPEL syntax without such standard attributes, BPELscript supports them. Thereby there are two main possibilities: (i) embed them in the code or (ii) reuse the annotations concept of Java. The first alternative is rejected since standard attributes are only optional and therefore it seems more sensible to not overload the basic syntax. Annotations on the other hand are meant "to associate information with the annotated program element" [Jav, 9.7]. Therefore, they provide a possibility to keep semantics through a translation if it is necessary.

Using Annotations for Attributes and in Basic Activities

After deciding to use annotations, it was clear to use the concept not only for standard attributes but for all attributes of BPEL not supported by the basic syntax. Currently, the attributes *queryLanguage* and *expressionLanguage* are supported only for the process itself. [5]

For annotations BPELscript uses the '@'-tag to start an annotation. An example of standard attributes in BPELscript is given in Listing 3.14 *b*. There are two main types of annotations: *boolean annotations* and *string annotations*. Since boolean annotations in BPEL have always the default value *no* it is sufficient to reflect the presence of the annotation in BPELscript with the *yes* value in BPEL. However, values of some attributes are inherited by all nested activities. As a result, it is necessary to have an overriding mechanism. Therefore boolean annotations can be negated by a *no* keyword after the annotation tag. In contrast to boolean annotations, string annotations have a mandatory string value which follows directly after the annotation tag.

[5] For all omitted attributes, it is possible to infer this information from the context. Therefore, the attributes have to be parsed by *Island Grammars* which can decide which language is used.

Listing 3.14 Handling of Standard Attributes for Basic Activities

```
# a - BPEL
  name="NCName"?
  suppressJoinFailure="yes|no"?

# b - BPELscript
  @name "NCName"
  @suppressJoinFailure no
  activity
```

3.4.2 Standard Elements for All Activities

BPEL provides two optional standard elements for each activity: the *source* and the *target*, which are shown in Listing 3.15. They are used to establish synchronization relationships through *links* [OAS07a, 10.2].

Listing 3.15 BPEL Definition of Standard Elements for All Activities

```
  <targets>?
     <joinCondition expressionLanguage="anyURI"?>?
        bool-expr
     </joinCondition>
     <target linkName="NCName" />+
  </targets>

  <sources>?
     <source linkName="NCName">+
        <transitionCondition expressionLanguage="anyURI"?>?
           bool-expr
        </transitionCondition>
     </source>
  </sources>
```

Modeling Alternatives for Standard Elements

BPEL enables both graph-oriented and block-structured programming. To enable graph-oriented programming within a block-structured approach, it has to cope with *links*. Thereby, the semantics of links is given through the *dead-path-elimination* ([OAS07a, LR00, CKLW03, KMWL08]). Within the development of SimPEL, the ODE proposed ideas to establish graph-oriented programming within a block-structured approach [BAR, LK]. A main goal of this ideas is to combine the *syntax* of the *fork/join* [Lea] parallelism concept with the *semantic* of BPEL. For the modeling of standard elements there are three solutions which are shown in Listing 3.16.

i use an extended goto syntax [LK]

ii introduce new activities for forking and joining [BAR]

iii use an activity-like syntax with activities for forking and joining

Option (i) is more close to BPEL. BPEL does not use explicit activities to fork and join, but uses properties of activities for forking and joining. In the serialization, these properties are realized as sub elements of the current activity. [6] Option (ii) is more close to usual programming. In contrast to add new syntax, new statements with a certain semantics are introduced. Option (iii) is a trade-off between the two options. On the one hand, links are properties of activities. Thus, it seems to be natural to reflect them also as properties of statements in BPELscript (e. g. annotations). On the other hand, such properties are a new construct in common programming languages and thus these constructs tend to raise the feeling of strangeness at programmers (cf. Section 3.4.4).

Listing 3.16 Example of Standard Elements Solutions

```
# solution (i)
  # signal part
  lauto: var risk = autoAssessor.approve(request) l1=[$risk = 'high']->lhuman, l2=[risk =
      'low']->lapp
  # join part
  [l1 and l2] lapp: customer.appoved();

# solution (ii)
  # signal part
  risk = autoAssessor.approve(request);
  signal(lhuman, [$risk = 'high']);
  signal(lapp, [risk = 'low']);
  # join part
  join(lhuman, lapp);
  customer.appoved();
```

After the main alternatives of modeling standard elements are introduced, the following sections discus whether and how they can be used. The goal is to get the same, unambiguous representation for both, basic and structured activities.

Handling of Standard Elements in Basic Activities

For the handling of standard elements in basic activities there are five approaches which are discussed in the following. Thereby, solution (ii) to (v) are derived from solution (i).

i introduce new activities, embedded in the basic syntax [BAR]

ii prefix standard elements

iii suffix standard elements

iv split standard elements

v annotate standard elements

Solution (i) introduce *join* and *signal* as statements and interpret them as *links*. Thereby, it match joins with its following activity and signals with its previous one as shown in Listing 3.17 *a*. Another way is to prefix or suffix a block of join and signals to the activity. The former (ii) is shown in Listing 3.17 *b*

[6] Note, that goto syntax should handled carefully [Dij68]

and the second one (iii) is shown in Listing 3.17 *c*. A variation of the last two is shown in Listing 3.17 *e*. Here the brackets are dropped and annotations are used instead (cf. 3.4.1). Furthermore, standard elements can be splitted such as shown in Listing 3.17 *d*.

Listing 3.17 Handling of Standard Elements

```
# a - introduce new activities
  join(...);
  activity;
  signal(...);
  signal(...);

# b - prefix attributes
  [join(...) signal(...) signal(...)] activity;

# c - suffix attributes
  activity [join(...) signal(...) signal(...)];

# d - split attributes
  [join(...)] activity; [signal(...) signal(...)]

# e - annotate attributes
  @join(...) @signal(...) @signal(...) activity;
```

Conceptually, all solutions are feasible, but the first solution is taken, since SimPEL and BPELscript addresses common programming languages. Thus, a clear and not overloaded syntax is provided for standard elements in basic activities.

3.4.3 Handling of Standard Attributes in Structured Activities

Regarding the handling of standard attributes in structured activities, the main problem is that they contain at least one *implicit sequence* (sequence handling in BPELscript is discussed later in Sections 3.5.1 and 4.3.6). The implicit sequence can have standard attributes on its own. This results in an ambiguity such as shown in Listing 3.18 *a*. As a result, it is insufficient to add an annotation in front of the structured activity since this leaks the usage of standard attributes for the containing sequence. Listing 3.18 *b* shows an idea which is discussed in detail in the next section with standard elements. Thereby annotations are allowed to occur after the preluding block structured keyword.

Listing 3.18 Handling of Standard Attributes for Structured Activities

```
# a - where the annotation belong? If or Sequence?
  @std_attr
  if (...) {...}

# b - idea: need annotations twice
  @if_std_attr
  if (...) @seq_std_attr {...}
```

3.4.4 Handling of Standard Elements in Structured Activities

For the handling of standard elements in block structured activities there are three approaches which differ in detail.

- status quo
- split blocking
- direct preluding

Listing 3.19 Handling of Standard Elements in Structured Activities

```
# a - status quo
  if (expr) join(...) { } signal(...), signal(...)
  else join(...) { } signal(...)

# b - split blocking
  if (expr) [join(...)] { } [signal(...) signal(...)]

# c - direct preluding
  if (expr) [join(...) signal(...) signal(...)] { }

# d - problem with status quo
  if (expr) join(...) { } signal(...), signal(...)
  else join(elseSequenceJoin) { } signal(elseSequenceSignal) signal(ifSignal);

# e - using both standard elements and attributes
  if (expr) [join(...) signal(...) signal(...)] std_attr? { }

# f - BPELscript
  if (expr) @join(...) @signal(...)* std_attr? { }
```

The first approach is to use standard elements as they are introduced in Section 3.4.1. It is shown in Listing 3.19 *a*. This alternative is rejected, because it introduces ambiguities which are shown in Listing 3.19 *d*. In this case there is an else-statement which contains a sequence with a join and a signal. The subsequent signal belongs to the if statement. The problem is to relate each signal to the correct statement or block. The semicolon suggests that both signals belong together, but it is unclear wherefore. In this case allowing a semicolon after a sequence signal must be also rejected due to the same reason. The second approach is to mark the standard elements of the implicit sequence using squared brackets. It is shown in Listing 3.19 *b*. This solution solves the ambiguity problem, but introduces a breakout since standard elements are used as statements *and* as split blocks. It is justifiable to break the standard behavior of standard elements, since they are rare in block structured activities. Nevertheless, this solution looks confusing and thus, it is also rejected. The third approach combines join and signal to place them in front of the containing block. It is shown in Listing 3.19 *c*. This solution looks superior, but is still disappointing when using standard attributes like in Listing 3.19 *e* since they are realized as annotations. To provide a clear concept BPELscript waives the idea of squared brackets and allows annotated standard elements for implicit sequences. This is shown in Listing 3.19 *f*.

In a nutshell, standard attributes in front of an activity belongs to the activity which follows directly after it. If the activity is structured and contains an implicit sequence, standard attributes for the implicit sequence are allowed like this is shown in Listing 3.19*f*.

BPEL allows to extend a process element with elements and attributes of namespaces other than those provided by BPEL itself [OAS07a, 5.1.2]. This is currently not supported by *bosto* and thus, should be provided by future work (refer to 5 on page 83).

3.4.5 Providing and Consuming Web Services—Receive, Reply, Invoke

BPEL provides a couple of simple facilities to enable sending and receiving messages from and to external partners (i. e. web services). These are the receive, reply and the invoke activity which are discussed in the following. The *receive* activity allows the business process to wait for messages [OAS07a, OAS07b, 5.2, 3.4.1]. The *reply* activity allows the business process to reply to a message. The *invoke* activity reflects the consumption of web services.

Modeling Alternatives for Functional Behavior

For modeling the functional behavior of processes there are two main possibilities which are shown in Listing 3.20 and discussed in the following [BAR, LK]:

i using preluding keywords

ii using a Java Script approach

As example the *receive* activity is used. To wait for messages from an external partner a receive activity has a mandatory *partnerLink* and an *operation* to specify the partners web service [OAS07a, OAS07b, 5.2, 3.4.1]. The requested data can be hold in a *variable* or in a set of *fromParts*. In detail in receive is discussed in Section 3.4.5.

Solution (i) is shown in Listing 3.20 *a.1*. It uses a preluding keyword with its (BPEL) attributes following in brackets. Optionally *fromParts* can be specified *or* the result can be stored in a *variable*. To indicate *parameter passing* modes for *fromParts*, the *with* keyword can be used. This leads to a variant shown in Listing 3.20 *a.2*. Solution (ii) is based upon the functional, imperative scripting language Java Script. It is shown in Listing 3.20 *b*. Thereby the partner link is treated as an object with the operation as method. The *fromParts* follows the operation in brackets.

To provide a clear and easy understandable language which is nearby original BPEL, both variants of solution (i) are preferred for modeling the functional behavior. Variant *a.2* is supported by BPELscript, since it reflects the parameter passing superior to variant *a.1*. After the general modeling style is set, the following subsections discuss how it is used to model the receive, reply and the invoke activity.

Listing 3.20 Syntax of Functions

```
# a.1 - using preluding keywords
  result = receive(partnerLink, operation, fromParts);

# a.2 - using preluding keywords
  result = receive(partnerLink, operation) with (fromParts);

# b - using a Java Script approach
  result = partnerLink.operation(fromParts);
```

Modeling Function Parameters—*from* and *to* parts

Data from a message partner can be copied directly to BPEL variables from an associated WSDL message variable. Similarly, data from BPEL variables can be copied to anonymous WSDL messages. [OAS07a, 10.4] "An alternative to the use of the *variable* attribute is the use of a collection of [*fromPart* or *toPart* elements]" [OAS07a, 12.7.1]. A chunk of the BPEL specification of the invoke activity is shown in Listing 3.21 *a*.

For the use of the *(out)variable* BPELscript uses the *assign* activity as illustrated in Listing 3.21 *b*. In case of multi-part WSDL messages, BPELscript provides the *with expression* containing a list of mappings from variables to parts. The *with* expression follows directly after the associated activity as it is shown in Listing 3.21 *c*. To gain a proper handling of from and to parts it was decided to adapt the parameter passing syntax from Ada 95 [Nag03]. Thereby an *in* part indicates *fromParts*, an *out* part indicates *toParts* and an *inout* part indicates both of them in one mapping. Thus, the listing shown in Listing 3.21 *c* results in the listing shown in part *d* of the listing. Note, that BPEL allows either a *variable* attribute or *fromPart*/*toPart* elements used on a activity. Currently, the necessary check is not provided by *bosto*.

Receive

The *receive* activity allows the business process to wait for a message from an external partner [OAS07a, OAS07b, 5.2, 3.4.1]. Therefore a receive activity has a mandatory *partnerLink* and an *operation* to specify the partners web service. For readability, a *portType* can be specified. The received data can be hold in a *variable*. With *createInstance* it is possible to create a new process instance and then consume an incoming message (value='yes') or to consume an incoming message by a running process instance (value='no'). The optional *messageExchange* attribute is used to provide a WSDL *request-response* operation by specifying an associated *reply* activity. To correlate incoming messages with running the business process instances *correlationSets* have to be specified. The usage of correlations is discussed in Section 3.7 on page 64. Finally, a list of *fromParts* can be specified. These can be regarded as function parameters of the receive activity. Listing 3.22 *a* provides an overview.

To model the receive activity BPELscript uses the preluding keyword approach. Listing 3.22 *b.1* provides a basic example and Listing 3.22 *b.2* presents an extended one. In addition, standard attributes, the receive statement offers the *portType*, *createInstance* and the *messageExchange* as optional annotations. For technical reasons the order of annotations is fixed following the order in the BPEL specification.

Listing 3.21 Translation of From and To Parts

```
# a - BPEL
  ...
  (out)variable="BPELVariableName"?
  ...
  <toParts>?
     <toPart part="NCName" fromVariable="BPELVariableName" />+
  </toParts>
  <fromParts>?
     <fromPart part="NCName" toVariable="BPELVariableName" />+
  </fromParts>

# b - BPELscript variables with an assignment
  variableName = invoke(...)

# c - BPELscript using from parts
  invoke(...) with (var1 : in NCName1, var2 : out NCName2, var3 : inout NCName3)

# d - BPEL result of c
  <invoke ...>
     <toParts>
       <toPart part="NCName2" fromVariable="var2" />
       <toPart part="NCName3" fromVariable="var3" />
     </toParts>
     <fromParts>
       <fromPart part="NCName1" fromVariable="var1" />
       <fromPart part="NCName3" fromVariable="var3" />
     </fromParts>
  </invoke>
```

Section 4.3.2 will seize this suggestion. The receive construct is structured as follows: The optional annotations precede the receive keyword. The receive takes the mandatory parameters *partnerLink*, *operation* and the optional *correlation*. Finally, the optional *fromParts* follows. The *fromParts* themselves are described above.

Reply

The *reply* activity allows the business process to reply to a message received by an associated inbound message activity (IMA). An IMA is a receive activity, an *onMessage* or an *onEvent* branch. Typically, the reply activity is used in conjunction with an IMA to form a WSDL request-response operation. [OAS07a, OAS07b, 5.2, 3.4.1] The reply construct is structured as follows: The optional annotations precede the reply keyword. The reply takes the mandatory parameters *partnerLink*, *operation* and the optional *correlation*. Finally, the optional *toParts* follows. The *toParts* themselves are described above.

In difference to the receive activity, the reply activity misses the *createInstance* attribute since it is not possible to create an instances within an reply. Instead, “a reply activity can come in two flavors: It can reply normal data [...] or it can reply faulted data [...]” [OAS07b, 3.4.1]. Thus it has a *faultName*

Listing 3.22 Translation of Receive

```
# a - BPEL
  <receive partnerLink="partner_NCName"
     portType="QName"?
     operation="operation_NCName"
     variable="BPELVariableName"?
     createInstance="yes|no"?
     messageExchange="NCName"?
     standard-attributes>
     standard-elements
     <correlations/>?
     <fromParts/>?
  </receive>

# b - BPELscript
# b.1
  variableName = receive(partner_NCName, operation_NCName);

# b.2
  @portType "QName"
  @createInstance
  @messageExchange "NCName"
  receive(partner_NCName, operation_NCName, correlations?) with (fromParts)?;
```

attribute. Furthermore, the "function parameters" of the reply activity get a different name: *toParts*, to reflect the different semantic of them. Listing 3.23 *a* provides an overview on it.

Listing 3.23 Translation of Reply

```
# a - BPEL
  <reply partnerLink="NCName"
     portType="QName"?
     operation="NCName"
     variable="BPELVariableName"?
     faultName="QName"?
     messageExchange="NCName"?
     standard-attributes>
     standard-elements
     <correlations/>?
     <toParts/>?
  </reply>

# b - BPELscript
# b.1
  reply(partner_NCName, operation_NCName, variableName);

# b.2
  @portType "QName"
  @faultName "QName"
  @messageExchange "NCName"
  reply(partner_NCName, operation_NCName, correlations?) with (toParts)?;
```

To model the reply activity BPELscript uses the preluding keyword approach. Listing 3.23 *b.1* provides a basic example and Listing 3.23 *b.2* presents an extended one. In addition, standard attributes, the reply statement offers the *portType*, *faultName* and the *messageExchange* as optional annotations. For technical reasons the order of annotations is fixed following the order in the BPEL specification. Section 4.3.2 will seize this suggestion. The reply construct is structured as follows: The optional annotations precede the reply keyword. The reply takes the mandatory parameters *partnerLink*, *operation* and the optional *correlation*. Finally, the optional *fromParts* follows. The *fromParts* themselves are described above.

Invoke

The *invoke* activity which allows the business process to invoke a *one-way* or *request-response* operation provided by a partner [OAS07a, OAS07b, 5.2, 3.4.1]. The one-way operation can be thought of an asynchronous procedure call, where the process continues with the process logic. The request-response operation can be thought of a synchronous function call, where at least a part of the process will be blocked "until it receives a response from the partner service" [OAS07b, 3.4.1]. In contrast to the previous activities, invoke has both an *inputVariable* (or its equivalent *toParts*) and an *outputVariable* (or its equivalent *fromParts*). The former is required by one-way invocation whereas the request-response invocation requires both. Since an invoke activity can be used as a shorthand notation of a scope activity [OAS07b, 3.4.1] it can have various handlers which are discussed in detail in Section 3.6 dealing with scopes on page 59.

To model the invoke activity BPELscript uses the preluding keyword approach too. Listing 3.24 *b.1* gives a basic example and *b.2* an extended one. According to BPEL the invoke activity provides the optional attributes *portType* and *standard attributes*. The main elements of invoke are the same as in receive and reply. In difference to them an invoke uses also the *inout* part mapping and has an optional compensation handler (which is discussed in detail in Section 3.6.2). To model the fault handling the invoke activity has to be enclosed by a *try* statement which will be translated directly into the form of Listing 3.24 *a*. Therefore, BPEL provides a shortcut containing an invoke in a scope. This shortcut is currently not supported by BPELscript, since it is (a) easier to use BPELscripts *try* statement instead and (b) does not increase readability. Listing 3.24 *b.3* gives a raw structure of fault handling within invoke.

3.4.6 Updating Variables and Partner Links—Assign

The previous sections discussed how business data can be received into a process. To split up this data in various parts the BPEL specification states, that "[the *assign*] activity can be used to copy data from one variable to another, as well as to construct and insert new data using expressions." [OAS07a, 8.4] The realization of the assign activity in BPELscript is discussed in Section 3.3.2 on page 33.

Listing 3.24 Translation of Invoke

```
# a - BPEL
  <invoke partnerLink="NCName"
     portType="QName"?
     operation="NCName"
     inputVariable="BPELVariableName"?
     outputVariable="BPELVariableName"?
     standard-attributes>
     standard-elements
     <correlations/>?
     <catch/>*
     <catchAll/>?
     <compensationHandler/>?
     <toParts/>?
     <fromParts/>?
  </invoke>

# b - BPELscript
# b.1
  variableName = invoke(partner_NCName, operation_NCName, variableName);

# b.2
  @portType "QName"
  invoke(partner_NCName, operation_NCName, correlations) with (fromToParts) compensation
      {...};

# b.3
  try {
    invoke(partner_NCName, operation_NCName, correlations);
  }
  catch {...}
```

3.4.7 Validating Variable Values—Validate

Values that are stored in variables can change during a process execution. Therefore, it cannot always be guaranteed that they are valid according to their definitions. Thus, BPEL provides the *validate activity* to validate values of variables against their associated XML and WSDL type definitions. Furthermore BPEL provides the *validate attribute* of the assign activity which is discussed in section 3.3.2 on page 33.

The validate activity has a *variables* attribute which accepts a list of variable names separated by whitespaces [OAS07a, 8.1]. The representation of the validate activity in BPELscript is using the preluding keyword concept with a list of variables following. The syntax of the validate activity is shown in Listing 3.25. Thereby, the *list_of_variables* is comma separated.

"A WS-BPEL implementation MAY provide a mechanism to turn on/off any explicit validation [...]" [OAS07a, 8.1]. Although it seems to be obvious to provide an automatic validation by the BPELscript parser[7], it was decided to keep this activity explicitly. The reason comes with additional resource

[7] This is not included in the current version of *bosto*, but is considered in chapter 5 on page 83.

Listing 3.25 Translation of Validate

```
# BPEL
  <validate variables="BPELVariableNames" standard-attributes>
     standard-elements
  </validate>

# BPELscript
  validate list_of_variables;
```

consumption of explicit variable validation at runtime of workflow engines. Thus it was necessary to keep this semantic during the translation.

3.4.8 Signaling Internal Faults—Throw

The *throw* activity is used to signal an internal fault explicitly. Since a fault must be identified with a QName the throw activity provides a mandatory *faultName* attribute and can provide additional information with the *faultVariable*. [OAS07a, 10.6] The throw activity can be handled with the preluding keyword concept. The optional attribute is realized with an annotation. The activity itself is used via the keyword *throw* followed by the QName of the fault to be thrown. The syntax is shown is Listing 3.26.

Listing 3.26 Translation of Throw

```
# BPEL
  <throw faultName="QName"
     faultVariable="BPELVariableName"?
     standard-attributes>
     standard-elements
  </throw>

# BPELscript
  @faultVariable "VariableName"
  throw (? ns_id )?;
```

3.4.9 Delayed Execution—Wait

There are situations, where a process has to wait for a specified time period, until a certain point in time is reached or specifies timer events for asynchronous execution. Therefore, BPEL provides three flavors of waiting semantics which are here described using a common denominator called *abstract-wait*. The three flavors are:

- delayed execution with the *standard wait activity*
- selective event processing with the *pick activity*
- concurrent event processing with the *eventHandler*

Although BPEL has no *abstract-wait*, the realization in BPELscript is conform to it.

The abstract-wait is shown in Listing 3.27 *a* and comes with optional standard attributes and elements. Furthermore it has an optional part for specifying the nature of waiting (wait *for* a duration or *until* a deadline). This part can only be optional in an *event handler*. Event handler have an optional *repeatEvery* part. When using the *repeatEvery* part, at least one expression must be specified. Finally, (a) a scope can follow if the activity appears in an *event handler*, (b) an activity if it appears in a *pick* and (c) nothing if it is a *standard wait activity*.

Listing 3.27 Translation of Wait

```
# a - BPEL
  <abstract-wait standard-attributes?>
     standard-elements?
     (<for expressionLanguage="anyURI"?>duration-expr</for>
     |
     <until expressionLanguage="anyURI"?>deadline-expr</until>)?
     <repeatEvery expressionLanguage="anyURI"?>
         duration-expr
     </repeatEvery>?
     scope-or-activity-or-nothing?
  </abstract-wait>

# b - BPELscript - (standard) wait for duration (<for>duration</for>)
  alarm (expr) scope?

# c - BPELscript - (standard) wait until deadline (<until>deadline</until>)
  timeout (expr) scope?

# d - BPELscript - busy wait (<repeatEvery>)
  repeatEvery (expr) scope?

# e - BPELscript - combined repeatEvery wait
  alarm (expr) repeatEvery (expr) scope
```

Delayed Execution with the Standard Wait Activity

For modeling the *wait activity* the abstract-wait is split into two statements:

- a durational wait named *alarm*
- a deadline wait named *timeout*

In case of a standard wait activity, BPELscript allows standard attributes and standard elements. However, it forbids the usage of an activity or scope in this context. Thus, for the standard wait BPELscript uses the statements *b* and *c* shown in Listing 3.27 *without using the scope*.

Selective Event Processing with the Pick Activity

For selective event processing the *pick activity* (cf. Section 3.5.4) allows only the waiting choice, *without standard attributes and elements*, followed by an activity. BPELscript uses the scope for modeling this, since further analysis phases can translate this into an single activity. Thus, for the "selective" wait BPELscript uses the statements *b* and *c* shown in Listing 3.27 *using the scope*.

Concurrent Event Processing with the Event Handler

Finally, the *eventHandler* can be modeled by expanding the pick modeling with the usage of the *repeatEvery* part. For *repeatEvery* BPELscript uses another keyword which is shown in Listing 3.27 *d*. Therefore it must be ensured (by the parser), that at least one of the possibilities is taken. Thus, for the wait inside an event handler, BPELscript uses statements *b*, *c* and *d*. Furthermore, the first followed by *d* can be used with the combined approach. An example of the combined *repeatEvery* is given in Listing 3.27 *e*.

3.4.10 Doing Nothing—Empty

The *empty* activity is a "no-op" in business processes. It is used for synchronization of concurrent activities and for fault handlers that consume a fault without acting on it. BPELscript uses *nop* as state,emt for the representation of the empty activity.

3.4.11 Adding new Activity Types—ExtensionActivity

BPEL allows the definition of new activity types, not defined by the specification, with the *extensionActivity*. Modeling this activity is hard since BPEL allows its own standard attributes and elements inside of the *anyElementQName* element. However, BPELscript provides a clear syntax for this as shown in Listing 3.28. In order to have a tense structure, extension activities are treated as normal statements which have no preluding keyword, but surrounded with triple curly brackets [LK]. Thus standard elements and attributes can be handled as they were discussed in sections 3.4.1 and 3.4.2. Technical details of necessary preprocessing are discussed in Section 4.2.1 on Page 77.

3.4.12 Immediately Ending a Process—Exit

The *exit* activity is used to immediately and explicitly end all currently activities in a process instance within which the *exit* activity is contained, "on all parallel branches, without involving any termination handling, fault handling, or compensation behavior" [OAS07b, 4.5]. The representation of the exit activity in BPELscript is straightforward, just using *exit;* as keyword.

Listing 3.28 Translation of Extension Activity

```
# BPEL
  <extensionActivity>
     <anyElementQName standard-attributes>
        standard-elements
     </anyElementQName>
  </extensionActivity>

# BPELscript
  {{{
     <anyElementQName/>
  }}}
```

3.4.13 Propagating Faults—Rethrow

The *rethrow* activity is used within fault handlers to rethrow the fault that was caught [OAS07a, 10.11]. Modifications to the fault data must be ignored by the *rethrow*. BPELscript models *rethrow* by using the rethrow keyword as presented in Listing 3.29.

Listing 3.29 Translation of Rethrow

```
# BPEL
  <rethrow standard-attributes>
     standard-elements
  </rethrow>

# BPELscript
  rethrow;
```

3.5 Structured Activities

The previous section gave an overview of the translation of the elemental steps of a process with basic activities. This section now describes the translation of structured activities which draw the order in which a collection of activities is executed.

Structured activities describe the order of control flow and contain other activities recursively. There are three main categories of control flow pattern: sequential structuring, parallel processing and selective event processing. The first might be known from traditional programming languages since it defines sequential control between activities. Examples for this category are the *sequence* and the *if* statement, further repetitive activities like *while*, *repeatUntil* and the serial variant of *forEach*. The second class is all about concurrency and synchronization using the *flow* activity and the parallel variant of *forEach*. The third category uses the *pick* activity to wait for the occurrence of internal and external events and then executes the activity associated with that event [OAS07a, 11].

This section discusses how these three categories are modeled in BPELscript. Thereby, the term *activity* is used to include both basic and structured activities.

3.5.1 Sequential Processing—Sequence

To define a collection of activities which have to be performed sequentially in lexical order the *sequence* activity is used [OAS07a, 5.2]. This is shown in Listing 3.30 *a*. In BPELscript everything is an implicit sequence and is enclosed by single curly brackets. Standard attributes and elements prefix the sequence as annotations as discussed in Section 3.4.1 and 3.4.2. In the case of a structured activity including a sequence, such as the *if* activity, the standard attributes and elements for the sequence appears after the keyword. Listing 3.30 *b* shows an example with the *if* activity which is discussed in the next section.

Listing 3.30 Translation of Sequence

```
# a - BPEL
  <sequence standard-attributes>
     standard-elements
     activity+
  </sequence>

# b - BPELscript sequence within if activity
  join(...);       #for the if statement
  @name "NCName" #for the if statement
  if (expr)
  @join(...)? @signal(...)* @name "NCName"? #for the sequence
  {...}
```

This modeling of sequences has the disadvantage that everything inside of a structured statement is an (implicit) sequence. For example, this is hideous when having a process with a *flow* activity as root element, as it is shown is Listing 3.31. A solution is to include a removal of sequences containing a single activity in further phases of a translation. As a pre-condition of this code-cleaning, the sequence may not have any incoming links. [8]

3.5.2 Conditional Behavior—If

To select exactly one activity for execution from a set of choices BPEL provides the *if* activity. Since sections 3.4.1 and 3.4.2 gave already a discussion about the main modeling alternatives of standard attributes and elements, the if activity is straightforward in BPELscript. The introducing keyword is followed by an expression, multiple *elseif* statements which can have their own condition expression and an optional concluding *else* statement. Listing 3.32 gives an example of the if activity in BPEL and BPELscript.

[8] This is not included in *bosto* yet. Cf. chapter 4.3.6 on 81.

Listing 3.31 Interpreting Everything as Sequence Leads to Excessive BPEL Code

```
# a - BPELscript
  process p {
    flow {nop;}
  }

# b - resulting BPEL
  <process name="p" ...>
    <sequence>
      <flow>
        <sequence>
          <empty/>
        </sequence>
      </flow>
    </sequence>
  </process>
```

Listing 3.32 Translation of If

```
# BPEL
  <if standard-attributes>
     standard-elements
     <condition expressionLanguage="anyURI"?>bool-expr</condition>
     activity
     <elseif/>*
     <else/>?
  </if>

# BPELscript
  if (bool-expr) {...}
  (elseif (bool-expr) {...} )*
  (else {...})?
```

3.5.3 Repetitive Execution—While, RepeatUntil, Serial ForEach

BPEL provides the serial *forEach* activity for expressing repetitive execution. The forEach activity provides also a variant for processing the included activity in parallel. Therefore, all variants of the forEach are described in Section 3.5.6.

While

To repeat a child activity as long as the specified condition is true BPEL provides the *while* activity. Just as the if activity the modeling of the while activity is straightforward in BPELscript. With preluding standard attributes the keyword *while* follows with the condition expression. In front of the loop body there can be optional standard elements as they were discussed before. An example of the while activity is shown in Listing 3.33.

Listing 3.33 Translation of While

```
# BPEL
  <while standard-attributes>
     standard-elements
     <condition expressionLanguage="anyURI"?>bool-expr</condition>
     activity
  </while>

# BPELscript
  while (bool-expr) {activity}
```

RepeatUntil

To repeat a child activity until a specified condition becomes true BPEL provides the *repeatUntil* activity. As well as the while activity the repeat until activity is straightforward, since only the looping paradigm changed from pre-test loop to post-test loop. Listing 3.34 gives an example of the repeat until activity.

Listing 3.34 Translation of Repeat Until

```
# BPEL
  <repeatUntil standard-attributes>
     standard-elements
     activity
     <condition expressionLanguage="anyURI"?>bool-expr</condition>
  </repeatUntil>

# BPELscript
  repeat {activity} until (bool-expr)
```

3.5.4 Selective Event Processing—Pick

The *pick* activity is used to trigger subsequent steps in a business process with messages by waiting for an arrival (*onMessage*) or for a duration and a deadline respectively (*onAlarm*). Each trigger is associated with a child activity. The pick activity completes when its child activity completes. It is shown in Listing 3.35.

The modeling of the pick activity is adopted from SimPEL [BAR]. It follows the preluding annotations and keyword approach, having a (sequence) body following. Within this body there are both, event based waiting with *onMessage* and periodical waiting with *onAlarm*, which are discussed in the following subsections. The *createInstance* attribute can be specified to cause a new process instance similar to the *receive* activity [OAS07b, 4.2.1].

Listing 3.35 Translation of Pick

```
# BPEL
  <pick createInstance="yes|no"? standard-attributes>
     standard-elements
     <onMessage partnerLink="NCName"
        portType="QName"?
        operation="NCName"
        variable="BPELVariableName"?
        messageExchange="NCName"?>+
        <correlations/>?
        <fromParts/>?
        activity
     </onMessage>
     <onAlarm>*
        (<for expressionLanguage="anyURI"?>duration-expr</for>
        |
        <until expressionLanguage="anyURI"?>deadline-expr</until>)
        activity
     </onAlarm>
  </pick>

# BPELscript
  @createInstance?
  std_attr?
  pick {
     onMessage+
     timeout*
  }
```

Waiting for a Message Arrival—onMessage

Since "each *onMessage* element points to a web service operation which is exposed by a business process and to a variable that holds the received message" [OAS07b, 4.2.1], the message activity is modeled with a *receive* activity which is extended by a parameterized block. A *parameterized block* in BPELscript is a simple block enclosed by curly brackets which has a parameter list enclosed by pipes at the beginning. [9] The first is shown in Listing 3.36 *a* and the second is shown in Listing 3.36 *b*. This modeling is equivalent since the receive activity has the same attributes and elements. Further it can be explicitly mapped to the message element.

Periodical Waiting—onAlarm

Each *onAlarm* element has a durational or a deadline wait expression following an activity which should be performed when the message or timeout occurs. The modeling of the *onAlarm* element follows the discussion of Section 3.4.9 and is shown already in Listing 3.27 *b* and *c*.

[9] This is not the full blown *receive*. No annotations are used and the handling of variable/from parts are done with the parameterized block.

Listing 3.36 Translation of onMessage

```
# a - BPELscript - onMessage
  onMessage : 'onMessage' '(' p=ID ',' o=ID (',' correlation)? ')' with_ex?
    param_block

# b - BPELscript - parameterized block
  param_block : '{' ('|' ID (',' ID)* '|')? proc_stmts+ '}'
```

3.5.5 Parallel and Control Dependencies Processing—Flow

To perform concurrent processing of activities BPEL provides the *flow* activity. Therein *links* can be specified to define explicit control dependencies between nested child activities. [OAS07a, 5.2] The keyword *flow* is a weak alternative to reflect the semantic of concurrent execution. Thus, SimPEL decided to provide *parallel* as keyword instead, since it sounds more familiar with concurrent execution. Listing 3.37 presents the modeling of BPEL's *flow* activity. At this, notice that *links* are realized in BPELscript by designated join and fork statements or by attributes (see Section 3.4.2).

Listing 3.37 Translation of Flow

```
# a - BPEL
  <flow standard-attributes>
     standard-elements
     <links/>?
     activity+
  </flow>

# b - BPELscript
  parallel {activity_1}
  ...
  (and {activity_n})*
```

3.5.6 Processing Multiple Branches—ForEach

The *forEach* activity iterates its child scope activity exactly N+1 times. Thereby N equals the *finalCounterValue* minus the *startCounterValue*. The *forEach* activity is serial in its default behavior and furthermore, it provides the possibility of parallel processing.

Instead of processing a sequential loop, it might make sense to start all loop iterations at the same time and process them in parallel [OAS07b, 4.3]. Therefor, the *forEach* loop provides, besides the usual standard attributes and elements, two attributes: the *counterName* and the boolean *parallel* tag. Furthermore, it has a *startCounterValue* (sCV), a *finalCounterValue* (fCV) and an optional *completionCondition* (cC) element followed by a child *scope*. The completion condition allows an early exit if it is smaller than the final counter value.

Since the completion condition is optional there might be two different kinds of modeling which are discussed in the following:

i the *combined approach* (Listing 3.38 *b*)

ii the *separated approach* (Listing 3.38 *c*).

The first is realized by BPELscript, the former by SimPEL.

The combined approach (i) uses a Java-like form of the *for* loop. A preluding keyword is followed by a set of parameters enclosed by brackets. The first parameter specifies the loop variable and the start expression, which are BPEL's counter name and start counter value. After the start expression follows the final counter expression and an optional completion expression. The attributes *parallel* of the *forEach* itself and *successfulBranchesOnly* of the *branches* element in the completion condition are handled with annotations. The second approach (ii) splits BPEL's *forEach* activity into two statements: *forAll* and *for*. The *forAll* iterates over all elements *from* a named start counter expression *to* the final counter expression. The *for* loop is similar to the combined approach using both counter values and the completion condition. Since BPELscript tries to simplify the main BPEL syntax it provides the combined approach, rejecting the separated approach.

Listing 3.38 Translation of For Each

```
# a - BPEL
  <forEach counterName="BPELVariableName" parallel="yes|no"
     standard-attributes>
     standard-elements
     <startCounterValue expressionLanguage="anyURI"?>
        unsigned-integer-expression
     </startCounterValue>
     <finalCounterValue expressionLanguage="anyURI"?>
        unsigned-integer-expression
     </finalCounterValue>
     <completionCondition>?
        <branches expressionLanguage="anyURI"?
           successfulBranchesOnly="yes|no"?>?
           unsigned-integer-expression
        </branches>
     </completionCondition>
     <scope/>
  </forEach>

# b - BPELscript - combined approach
  @parallel?
  @successfulBranchesOnly?
  for (ID = sCV; fCV?; cC?) scope

# c - SimPEL - separated approach
  forall (ID = sCV to fCV) scope
  for (ID = sCV; fCV; cC) scope
```

3.6 Scopes

The previous section gave an overview of structured activities in BPELscript. Thereby it discussed the three main categories of structured activities which are: sequential structuring, concurrent processing and selective event processing. This section now introduces *scopes* to define nested activities. Scopes extend the concept of nested blocks known from traditional programming languages.

Besides the global process container BPEL allows a hierarchy of nested *scopes*, which are known from traditional programming languages as *blocks*, to hide process data outside of that scope. Scopes are used like regular activities to control global data access. They can have its own entities like variable definitions, partner links, message exchanges, correlation sets, and handlers. Isolation semantics is indicated by the isolated attribute. Isolation semantics provides mutual exclusion. [OAS07b, 3, 4] If the process must exit immediately on a BPEL standard fault other then *bpel:joinFailure* the *exitOnStandardFault* attribute can be used [OAS07a, 5.2].

Listing 3.39 shows the modeling in BPELscript where scopes are intended to look as structured activities using a preluding scope keyword [BAR]. Thereby, having the name attribute as optional identifier and the body following in curly brackets. Afterwards follows activities as they were introduced in the previous sections. Furthermore, BPELscript provides a specific handling of scopes which are embedded in structured activities or in the *onEvent* activity discussed later in this section. Since it is uncomfortable to always write the keyword *scope*, it was decided to skip it in certain cases. Thus BPELscript provides a short hand for embedded scopes which are only allowed in *onEvent* (See Section 3.6.5), *onAlarm* (See Section 3.5.4) and *forEach* (See Section 3.5.6).

The following subsections describe the handling of *messageExchanges*, *Compensation*, *Faults*, *Termination* and *Events*.

Listing 3.39 Translation of Scopes

```
# BPEL
  <scope isolated="yes|no"? exitOnStandardFault="yes|no"?
     standard-attributes>
     standard-elements
     <partnerLinks/>?
     <messageExchanges/>?
     <variables/>?
     <correlationSets/>?
     <faultHandlers/>?
     <compensationHandler/>?
     <terminationHandler/>?
     <eventHandlers/>?
     activity
  </scope>

# BPELscript
  scope (ID)? {
    activity+
  }
  compensationHdl? terminationHdl? eventHdl?
```

3.6.1 Message Exchange Handling in BPELscript

If an execution can result in multiple IMA-reply pairs the optional *messageExchange* attribute must be used to disambiguate the relationships between them [OAS07a, 10.4.1]. In BPELscript the explicit definition of message exchanges orientates at the handling of variables, using a comma separated list of identifier with *messages* keyword.

3.6.2 Compensate Done Work—Compensation Handling in BPELscript

With compensation handling BPEL provides the capability of compensation spheres. This means, that compensation activities can be associated with activities to process repair actions. For more details on compensation spheres which are based on the *Saga* transaction concept, refer to [LR00] and for transaction processing in general to [GR93].

Compensation Handler

In BPEL and BPELscript only *scopes* can have compensation handlers. The compensation handler in BPELscript follows directly after the scope body [BAR]. [10] Syntactically, they are introduced via the common preluding keyword approach using *compensation* as keyword with an activity body following as illustrated in Listing 3.40.

Listing 3.40 Translation of Compensation Handlers

```
# BPEL
  <compensationHandler>?
     activity
  </compensationHandler>

# BPELscript
  compensation {
     activity
  }
```

Invoking Compensation Handler

Compensation handlers can be either invoked by the *compensate* or *compensateScope* activity. The *compensate* activity is used to refer to all inner scopes that have already completed successfully, in default order. [OAS07b, OAS07a, 4.1.4, 5.2] And "[the *compensateScope* activity activates] the compensation handler of one specified successfully completed scope to be executed." [OAS07b, 4.1.4] In BPELscript both are modeled with the same statement as shown in Listing 3.41.

[10] Currently the order is fixed to the one given in Listing 3.39. This should be resolved in future work.

Listing 3.41 Translation of Compensate and CompensateScope

```
# BPEL compensate
  <compensate standard-attributes>
     standard-elements
  </compensate>

# BPEL compensateScope
  <compensateScope target="NCName" standard-attributes>
     standard-elements
  </compensateScope>

# BPELscript
  compensate (NCName)?
```

3.6.3 Undoing Completed Work—Fault Handling in BPELscript

To do corrective actions in case of a fault BPEL provides fault handling to switch from normal processing. Since different faults can require different fault handling BPEL provides two types of reaction: catch a single fault referred by a *faultName* with the *catch* activity or *catchAll* faults with a wildcard-like error handler. In BPELscript fault handlers only occur in conjunction with the *try* statement as this denoted in Section 3.4.5 on page 47 [BAR]. Since the *try* statement does not exists in BPEL it will add the declared *catch* expressions to the *faultHandler* of its enclosing *scope*. For a better understanding Listing 3.42 shows the syntax of fault handlers. The optional attributes *faultName* and *faultVariable* are directly provided by the basic syntax whereas the *faultMessageType* and *faultElement* are modeled with annotations.

Listing 3.42 Translating Fault Handlers

```
# BPEL
  <faultHandlers>
     <catch faultName="QName"?
        faultVariable="BPELVariableName"?
        ( faultMessageType="QName" | faultElement="QName" )? >*
        activity
     </catch>
     <catchAll>?
        activity
   </catchAll>
  </faultHandlers>

# BPELscript
  try {
     activity
  }
  annotations?
  catch ( ns_id ) { |faultVar|?
     activity
  }
  catchAll {...}
```

Listing 3.43 shows a translation of a *try* statement. The example illustrates that the body of the try statement appears at the position of the try statement itself, whereas the try statement will disappear. The catch statements are translated into catch activities.

Listing 3.43 Example: Usage of Fault Handlers

```
# a - BPELscript
  scope {
     exit;
     try {nop;}
     catch (Exception) {|exc| exit;}
  }

# b - BPEL
  <scope>
    <faultHandlers>
      <catch faultName="Exception"
           faultVariable="exc">
        <exit/>
      </catch>
    </faultHandlers>

    <sequence>
       <exit/>
       <empty/>
    </sequence>
  </scope>
```

3.6.4 Terminating Running Work—Termination Handling in BPELscript

If a fault occurs in a scope, all running activities within an associated scope have to be terminated. Thereby, structured activity behavior is interrupted by propagating the termination into the contained activities. The activities *assign*, *empty*, *throw*, *rethrow* and *exit* can complete while others are interrupted. [OAS07b, OAS07a, 4.1.3, 12.6] Since it may necessary that an activity has to clean up before terminating, BPEL provides *terminationHandler* which can be thought of as “last wishes”. The only way to use termination handler is within scopes as denoted in Listing 3.39. BPELscript uses the preluding keyword with *onTermination* for termination handling. Listing 3.44 illustrates this.

Listing 3.44 Translating Termination Handlers

```
# BPEL
  <terminationHandler>
     activity
  </terminationHandler>

# BPELscript
  onTermination {
     activity
  }
```

3.6.5 Event Handling in BPELscript

In business processes it is important to wait at some point on an *event* [LR00, p. 185]. Since it is not always desired to interrupt the business process logic with a synchronous wait or a blocking receive, BPEL provides an asynchronous execution with *eventHandlers*. They are associated with its enclosing process (or scope) whose lifecycle determines the lifecycle of the handler. *EventHandlers* are invoked when the corresponding event occurs. There are two types of events: inbound messages corresponding to a WSDL operation or alarm going off after user-set times. [OAS07a, OAS07b, 12.7, 4.2.3] The BPEL and BPELscript grammars of event handlers are illustrated in Listing 3.45. In the following only the *onEvent* part is described since waiting semantics is already discussed in section 3.4.9 on page 49.

Listing 3.45 Translating Event Handlers

```
# BPEL
  <eventHandlers>?
     <onEvent partnerLink="NCName"
        portType="QName"?
        operation="NCName"
        ( messageType="QName" | element="QName" )?
        variable="BPELVariableName"?
        messageExchange="NCName"?>*
        <correlations/>?
        <fromParts/>?
        <scope/>
     </onEvent>
     <onAlarm>*
        (<for expressionLanguage="anyURI"?>duration-expr</for>
        |
        <until expressionLanguage="anyURI"?>deadline-expr</until>)?
        <repeatEvery expressionLanguage="anyURI"?>?
           duration-expr
        </repeatEvery>
        <scope/>
     </onAlarm>
  </eventHandlers>

# BPELscript
  events { onEvent* onAlarm* }
```

Inbound Messages corresponding to WSDL Operations

The handling of inbound messages corresponding to WSDL operations is similar to the handling of receiving messages. Thus this section provides a short summary of the syntax shown in Listing 3.46, referring to the detailed discussion in Section 3.4.5. An event can be annotated with a *portType*, *messageType* or *element* (there are no standard elements in this context). The event keyword itself follows the partner link, operation and correlation, furthermore events can either use a single variable or its equivalent *fromParts* as discussed earlier.

Listing 3.46 Syntax of Event Handlers

```
# BPELscript
  variableName = event (partnerLink, operation (, correlation)? ) {...}

# or using fromParts
  event (partnerLink, operation (, correlation)? ) with (fromParts) {...}
```

3.7 Correlation

A business process can have multiple conversations with partners where messages to stateful business processes need to be delivered to the correct destination port *and* to the correct instance of the process that provides the port. Since the use of stateful interaction references is insufficient in this case, BPEL provides a declarative approach to specify the structure and position of correlation tokens that are used in *correlationSets*. Thus, infrastructure can provide instance routing automatically. [OAS07a, 9.1]

This section discusses modeling alternatives of correlations by means of BPEL, SimPEL and BPELscript. Afterwards the declaration and usage of correlation in BPELscript is introduced.

3.7.1 Message Correlation in BPEL, SimPEL and BPELscript

For the discussion of message correlation a part of the auction service from [OAS07a, 15.4.2] is shown in Listing 3.47 and compared at first with the SimPEL approach.

The example shows two receive activities using the same correlation set. Listing 3.48 illustrates the equivalent example in SimPEL where the indirection and overhead is reduced to a bare minimum to provide a summary of how to extract the data from a message and which process data should be mapped to it [Rio08]. The approach has two disadvantages, (a) no join pattern support and (b) no grammar support. First of all, join pattern are not supported in SimPEL. Second, the proposed approach leads to a modified semantics of "receive". The presented handling uses the *receive* activity similar to the *onMessage* activity introduced in Section 3.5.4 on Page 55. In addition, the handling uses a semantically different syntax not provided by the grammar at all. Furthermore, it summarizes the two independent receive activities to a single one using an *if* activity inside. This summarization should be left to optimization phases since it requires a broad semantical understanding. A statical analysis would be a better approach than embedding it directly in a translation. For more detail on *static analysis* refer to [NNC05].

Since efficiency should subordinate good structure [Bal05, p. 414], BPELscript went a different way of modeling correlations, orienting itself on BPEL. Therefore *correlationSets* are modeled using the *correlates* keyword which can be followed by semicolon separated *correlationSets*.

For completeness, Listing 3.50 shows the translation of *correlationSets* in BPELscript.

Listing 3.47 Part of Auction Service

```
<correlationSets>
   <correlationSet name="auctionIdentification"
     properties="as:auctionId" />
</correlationSets>
...
<flow>
   <!-- Process seller request -->
   <receive name="acceptSellerInformation"
       partnerLink="seller"
       portType="as:sellerPT"
       operation="submit"
       variable="sellerData"
       createInstance="yes">
       <correlations>
          <correlation set="auctionIdentification" initiate="join" />
       </correlations>
   </receive>

   <!-- Process buyer request -->
   <receive name="acceptBuyerInformation"
       partnerLink="buyer"
       portType="as:buyerPT"
       operation="submit"
       variable="buyerData"
       createInstance="yes">
       <correlations>
          <correlation set="auctionIdentification" initiate="join" />
       </correlations>
   </receive>
</flow>
```

3.7.2 Declaring and Using Correlation Sets in BPELscript

BPEL provides a correlation consistency constraint for the initiate values of correlation sets. This means, that the initial values are constant. Thus, there are three types of legal values. *Yes* forces the activity to attempt the initialization, *join* initiates the correlation set if it is not yet initiated and the default value *no* lets the activity must not attempt the initialization. [OAS07a, 9.2] In BPELscript, correlations can be used within the parameter list of its associated activity using curly brackets. The solution results in the way initiate values are used—either *default*, *join* or *force*. Thus, BPELscript supports to use keywords in front of each correlation set like this is shown in Listing 3.51 (default values have no keyword at all). For force and join semantics BPELscript provides also a short cut, using the exclamation point (!) for force and question mark (?) for join.

In case of an synchronous invoke activity as presented in Section 3.4.5 on Page 47, BPEL provides a pattern attribute on correlation specification. The pattern is used to indicate either an outbound message (*request*), an inbound message (*response*) or both (*request-response*). [OAS07a, 9.2] In BPELscript, these patterns are modeled similar to initiate values explained before. For an outbound message it uses

Listing 3.48 Part of Auction Service in SimPEL [ODEa]

```
function auctionIdFromBuyer(buyer) {
  buyer.ID;
}
function auctionIdFromSeller(seller) {
  seller.auctionId;
}

process Auction {
# The original BPEL spec example for the auctions process uses a join pattern which
# is not supported in SimPEL.
receive(buyerOrSeller, submit) { |startData|
  if (startData.buyer) {
    buyerData = startData;
    auctionId = auctionIdFromBuyer(buyerData);
    sellerData = receive(seller, submit, {auctionIdFromSeller: auctionId});
  } else {
    sellerData = startData;
    auctionId = auctionIdFromSeller(sellerData);
    buyerData = receive(buyer, submit, {auctionIdFromBuyer: auctionId});
  }
}
```

Listing 3.49 Part of Auction Service in BPELscript

```
process Auction {

correlates { auctionIdentification(auctionId); }

flow {
  @name "acceptSellerInformation"
  sellerData = receive(seller, submit, {?auctionIdentification});
  @name "acceptBuyerInformation"
  buyerData = receive(buyer, submit, {?auctionIdentification});
}}
```

Listing 3.50 Translating CorrelationSets

```
# BPEL
  <correlationSets>?
     <correlationSet name="NCName" properties="QName-list" />+
  </correlationSets>

# BPELscript
  correlates { corr_set* } #semicolon separated

  corr_set : name ( properties* ); #properties - comma separated
```

the *request* keyword or @> as shortcut. Analog to it *response* (@<) and *request-response* (@<>). [11] Listing 3.52 shows the modeling of patterns.

[11] Conceptually it was planed to use the shortcuts without the @ symbol, but because of technical problems with ANTLR, it was decided to use @ for simplicity.

Listing 3.51 Solutions for Correlation Modeling—Initiate

```
# BPELscript
  sellerData = receive(seller, submit, {auctionIdentification});
  sellerData = receive(seller, submit, {force auctionIdentification});
  sellerData = receive(seller, submit, {join auctionIdentification});
```

Listing 3.52 Solutions of Correlation Modeling—Pattern

```
# using correlation patterns, long variant
  invoke(seller, answer, sellerAnswerData, {request auctionIdentification});
  invoke(seller, answer, sellerAnswerData, {response auctionIdentification});
  invoke(seller, answer, sellerAnswerData, {request-response auctionIdentification});
```

3.8 Document Linking

BPELscript provides a translation for BPELs *import* activity to support the import of WSDL and XSD formally [OAS07b, OAS07a]. Therefore the *import* statement contains two attributes. The first defines the namespace to import. The second is specifying the location using the *location* keyword. Imports are defined in the header of a BPELscript process file as shown in Listing 3.53. [12]

Listing 3.53 Translation of Import

```
# BPEL
  <import namespace="anyURI"?
     location="anyURI"?
     importType="anyURI"/>*

# BPELscript
  namespace ns_id = "anyURI";
  ...
  @type="anyURI"
  import id = ns_id::"location";
```

This chapter introduced all elements of a translation function from BPEL to BPELscript and vice versa, focusing on the semantical aspects. The next chapter explains , how this translation is realized technically.

[12] Type annotation can be removed if BPELscript has a type inference mechanism.

Chapter 4

Project Description of *bosto*: The BPEL to BPELscript Translator

The previous chapter introduced the translation from BPEL to BPELscript and vice versa for semantic purposes. This chapter dives into the technical aspects and present a translator named *bosto*—the BPEL to BPELscript Translator. The cornerstone of *bosto* is to use automated tools as they are introduced in Chapter 2: ANTLR, ANTXR, StringTemplate and gUnit. Thus, *bosto* provides a extensible and fault-tolerant system which can be easily changed.

In the following this chapter provides a theoretical solution and discusses how the solution is realized in detail. Afterwards some additional work is illustrated.

4.1 Project Architecture

In the real world, there is a gap between abstract design and technical implementation. This leads to a difference between the theoretical solution and the resulting implementation in *bosto* too. The gap is mainly caused by technical incompleteness of the used tools. This does not mean that the tools are unusable, erroneous or incomplete—they are great and powerful, but still in evolution. Hence, this section first introduces a theoretical solution and then describe how it is realized.

4.1.1 Direct in Place Translation

Since projects in information technology should try to keep things simple and stupid (*KISS* [Lam83]) the first question is why a direct translation, using a text emitting version of *grep* [1] for example, is not the solution. The answer is quite easy and comes with assistance from theoretical computer science: regular expressions comes with the strength of regular languages which are not powerful enough to handle semantic context. Since context is essential in a translation of programming languages it is necessary to process a highly condensed version of the input—the ASTs. Furthermore there is another

[1] global regular expression print—a command line text search utility [Mag00]

advantage when using ASTs. Translators will reach a high level of abstraction where further analysis phases and optimization phases can attach. [2]

4.1.2 Theoretical Solution

Since a translation needs a broad understanding of the translation artefacts, the main architecture of the theoretical solution is shown in Figure 4.1. In the process exists two branches—the BPEL-parsing part and the BPELscript-parsing part—which are connected via a tree translation between both ASTs.

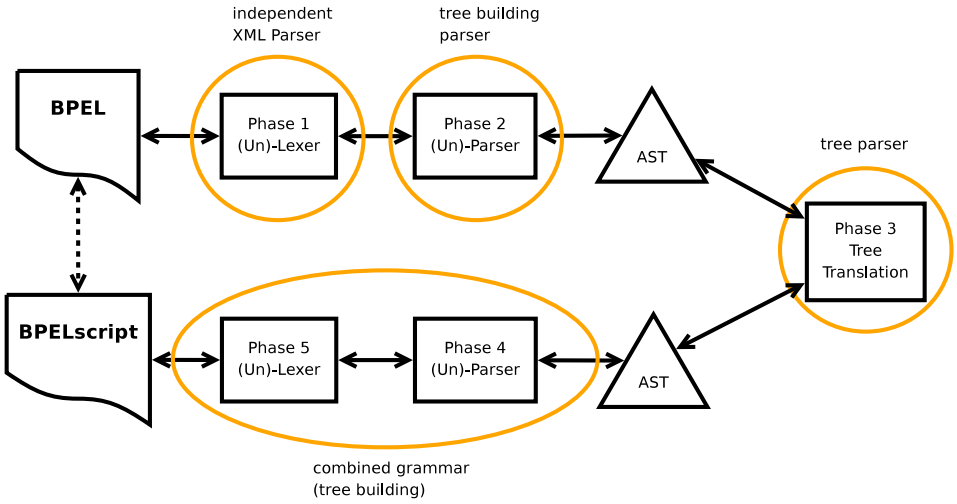

Figure 4.1: Bosto Translation Process—theoretically

Assuming that there exists a parser generator that can generate a parser as well as a "unparser"—which can automatically emit output from an AST—the translation splits into three parts:

1 process an AST_a

2 translate it into its counterpart AST_b

3 "unparse" the AST_b

The outcome of this are three questions which are explained in the following sections:

1 How to parse BPEL and BPELscript into ASTs?

2 How to do the tree translation?

3 How to do the unparsing?

To parse BPEL, this work uses ANTXR (see Section 2.3) with an independent XML parser and a blocking queue [SE]. XML is semi-structured and hence an AST is not used for this purpose. The unparsing is realized with embedded actions in the grammar. Parsing BPELscript is a classical

[2] For example, the *SSA* (static single assignment form [CFR+91]) is based on ASTs. SSA is used for analysis and optimization in compiler technology. In its interprocedural variant (ISSA [Bis08, SGSE07]), it can be also used for programs of the real world.

translation process using a combined ANTLR grammar for generating the lexer and the parser (see Section 2.2). From the parser grammar, a tree walker is derived to handle the translation. Finally, StringTemplates are used by the tree walker to generate the output.

Having illustrated the theoretical solution, the following sections illustrate how it can be realized. Thereby, the translation is cut into two parts: first starting with the translation of BPEL files and second translating BPELscript files back to BPEL.

4.1.3 Translating BPEL to BPELscript

Since BPEL is an XML language, the translation of BPEL has to cope with XML-parsing first. Therefore *bosto* forces extensibility and modularization by using an existing XML parser for lexing. As explained in Section 2.3, this can be SAX, DOM or any kind of XML parser. This XML parser feeds the *BlockingQueue* so that ANTXR, as "pull" API, can merge the *XMLTokenStream* by supplying a blocking queue [SE]. Thus, *bosto* can provide a simple grammar for parsing BPEL by just using XML-tags as non-terminals. The overall information flow is shown in Figure 4.2.

The preprocessing is necessary because of extension activities and is discussed in Section 4.2.1.

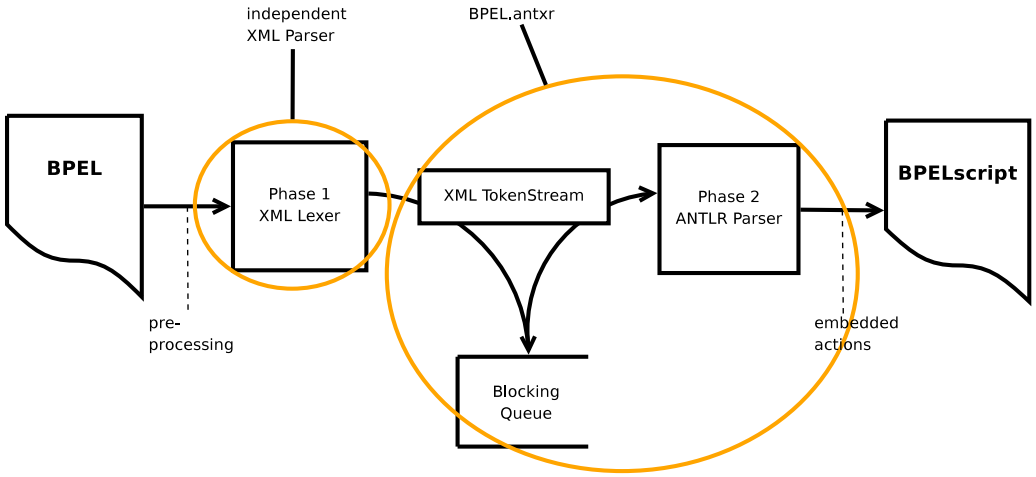

Figure 4.2: Bosto Translation Process—BPEL to BPELscript

4.1.4 Translating BPELscript to BPEL

Translating BPELscript to BPEL is a classical translation process using a combined ANTLR grammar for generating the lexer and the parser. Thereby the parser is enriched with ANTLRs tree rewriting rules to construct an AST. After the parser has created the AST, the tree parser/walker can walk over the tree to process the information via template calls. The overall information flow is shown in Figure 4.3.

The postprocessing of extension activities is discussed in Section 4.2.1.

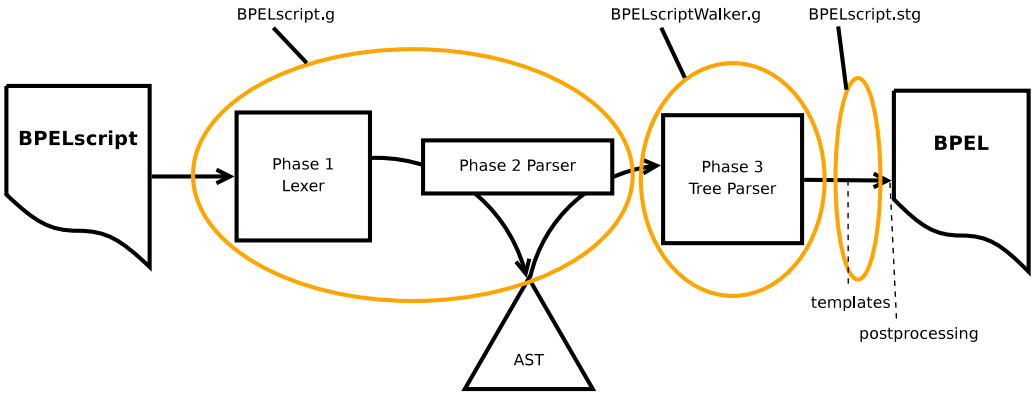

Figure 4.3: Bosto Translation Process—BPELscript to BPEL

4.1.5 An Exemplary Translation

This section provides an example of the translation of the pick activity to illustrate the presented steps. The explanation starts with the BPEL specification of *pick* and maps it to an ANTXR representation to generate BPELscript. To retranslate BPELscript, the ANTLR representation of *pick* is shown as combined grammar, tree grammar and its StringTemplate. The complete process translation is shown in Listing 4.1.

Part *a* of the listing defines *pick* as BPEL activity with two optional attributes *createInstance* and *standard-attributes*. Furthermore, the pick activity contains *standard elements*, at least one occurrence of *onMessage* and zero or more occurrences of *onAlarm*. Since standard attributes and elements are used by every activity, they are handled centralized in an own representation. Also *onMessage* and *onAlarm* are handled in an own representation.

Thus, ANTXR models the *pick* activity by using <pick> as non-terminal grammar rule which returns its content as a string. Listing 4.1 *b* shows that the ANTXR representation of pick consist of three parts:

- header part
- middle part
- action part.

In the *header part*, the attribute references, variables and constructing commands are placed. For *pick*, these are the attributes initially referenced by the @ operator. Furthermore, string variables are defined to represent the contents of standard elements, *onMessage* and *onAlarm*. In the *middle part*, the rule definition follows—for *pick* these are standard elements, *onMessage* and *onAlarm* again. Finally, the *action part* follows with the actions to perform within the *pick* activity—this is the handling of its attributes in front of the handling of the *pick* content. In addition, Listing 4.1 *b* illustrates why embedded actions are insufficient. Embedded actions as introduced in Sections 2.2.2 and 2.3.3 are surrounded by curly brackets. Without a template concept, they are distributed over the grammar to emit output. In Listing 4.1 *b*, this is insignificant since the actions are less complex. But with increasing complexity, the

actions overload the parser grammar with semantic details and leads to an confusing grammar which is hard to maintain.

The action-part results directly from the BPELscript representation shown in Listing 4.1 *c*. The pick statement can be preceded by the *createInstance* annotation or standard attributes. The *onMessage* and *onAlarm* content of *pick* is enclosed in curly brackets while the standard elements are derived in the tree grammar.

The ANTLR combined grammar representation also results directly from the BPELscript representation and is shown in Listing 4.1 *d*. As before, a *pick* starts with the optional attribute *createInstance* and standard attributes. They are followed by the *pick* keyword and its curly brackets and contain non-empty occurrences of *onMessage* and multiple occurrences of *onAlarm* statements. After the syntactic elements of *pick* are handled, a rewrite rule for the AST construction follows with the arrow operator. The rewrite rule is straightforward: the elevated wedge creates a tree node with all the pick information.

Next, Listing 4.1 *e* shows how standard elements are derived via parameter passing of complete statements in the AST. The rewrite rule from the combined grammar moves to the recognition part and a template call is defined via the arrow operator. To avoid infinite recursion the actual parameters must be explicitly assigned to its formal ones. [3]

Finally, Listing 4.1 *f* shows the template according to the *pick* statement. The template contains a template call to the *tagging* template, which handles the XML-tagging. Thereby, the caller can specify a *tagname*, *content* (these are XML-children) and *incontent* (these are XML-attributes). Comparing embedded actions to StringTemplate, it is clear that StringTemplate should be the favored solution.

The ensure correctness of the translation, the next section explains how it is tested using gUnit.

[3] Formal parameters are those, which are “formally” defined in the template definition. Actual parameters are those, which are given within the template call.

Listing 4.1 Complete Translation of Pick Activity

```
# a - BPEL
  <pick createInstance="yes|no"? standard-attributes>
     standard-elements
     <onMessage/>+
     <onAlarm/>*
  </pick>

# b - ANTXR
<pick> returns [String text=""]
     { //header
       BpelBoolean crtInst = BpelBoolean.toBpelBoolean(@createInstance);
       //standard attributes
       String name = @name; BpelBoolean dpe = BpelBoolean.toBpelBoolean(@suppressJoinFailure);
       String se=""; String onM=""; String oA="";
     }
     : se=std_elts (onM=<onMessage>[onM])+ (oA=<onAlarm>[oA])* //middle
     { //footer
       text+=//handle crtInstance and standard attributes
       text+="pick {\n"+ onM +"\n"+ oA +"}\n";
     };

# c - BPELscript
  @createInstance?
  std_attr?
  pick {
     onMessage+
     timeout*
  }

# d - ANTLR Combined Grammar
pick
     : CREATE_INST? std_attr         //handle annotations
     'pick' '{' onMessage+ onAlarm* '}'     //handle pick
      -> ^(PICK onMessage+ onAlarm* CREATE_INST? std_attr); //tree construction

# e - ANTLR Tree Grammar
pick [List join, List signal] //get standard elements if present
     : ^(PICK om+=onMessage+ to+=onAlarm* CREATE_INST? std_attr) //handle pick
     -> pick(oms={$om}, onalarm={$to}, join={$join}, signal={$signal},
          crt_inst={$CREATE_INST}, std_attr={$std_attr.st}); //template call

# f - StringTemplate
pick(oms, onalarm, join, signal, crt_inst, std_attr) ::=
<<
<taggin(tagname="pick",
        content=std_aux(join=join, signal=signal)
           +{<oms:{mes| <mes>}; separator="\n"><if (oms)><\n><endif>}
           +{<onalarm:{to| <to>}; separator="\n">},
        incontent={<if(crt_inst)><crt_inst><endif><if(std_attr)><\n> <std_attr><endif>}
       )><\n>
>>
```

4.1.6 Testcases

As denoted in Section 2.5, *bosto* uses gUnit to provide an easy and automated mechanism to ensure correctness of the translation. Therefore it is necessary to define a testsuite for each part of the translation. An example of such a testsuite is shown in Listing 4.2. Thereby, a testsuite starts by specifying the parser or tree parser which is to test. The main content of a testsuite is its testing specification which follows the syntax given in the listing. [4]

Listing 4.2 gUnit Testsuite Definition

```
// using ANTLR
gunit BPELscriptWalker walks BPELscript;

// syntax: 'grammar-rule-name' walks 'walker-rule-name' : 'file' -> 'directory-in testcases'
pick walks pick : pick -> "activities/"

// anomaly with ANTXR
__xml_pick : pick -> "activities/"
```

For the running *pick* example, the following has to be specified: the grammar-rule *pick walks* the walker-rule *pick* with the testfiles *pick.bpel* and *pick.bpelscript* found in the *activities/* directory. Examples how the testfiles looks like are shown is Listing 4.4. Note the anomaly for the ANTXR testing rules. Thereby, an additional prefix *__xml_* has to be set for XML-non-terminals. After the definition of the testsuite, gUnit can generate a jUnit testsuite. A part of the resulting testsuite, illustrating the running example part, is shown in Listing 4.3.

Listing 4.3 gUnit Generated jUnit Testsuite

```
public void test__xml_pick1() throws Exception {
   // test input: "pick"
   Object retval = execParser("__xml_pick", prefix+"activities/"+"pick"+".bpel", true);
   Object actual = examineParserExecResult(8, retval).toString().trim();
   Object expecting = null;
   expecting = readOutput(prefix+"activities/"+"pick"+".bpelscript");

   assertEquals("testing rule "+"__xml_pick", expecting, actual);
}
```

[4] For simplicity the path to the testcase directory is set to *"src/org/apache/ode/testcases/"* statically in the *junit.stg*. A direct path setting inside of the testsuite specification would be more usable. However, it is not provided within the gUnit modification.

Listing 4.4 gUnit Testfiles—pick

```
# pick.bpelscript
pick {
  @portType "orderEntry"
  onMessage (buyer, inputLineItem) { |lineItem|
    nop;
  }

  @portType "orderEntry"
  onMessage (buyer, orderComplete) { |completionDetail|
    nop;
  }

  alarm (['P3DT10H']) {
    join(setMessage-to-reply, approval-to-reply);
    @portType "lns::loanServicePT"
    reply(customer, request, approval);
  }
}

# pick.bpel
<pick>
   <onMessage partnerLink="buyer"
      portType="orderEntry"
      operation="inputLineItem"
      variable="lineItem">
      <empty />
   </onMessage>
   <onMessage partnerLink="buyer"
      portType="orderEntry"
      operation="orderComplete"
      variable="completionDetail">
        <empty />
   </onMessage>
   <onAlarm>
      <for>'P3DT10H'</for>
       <scope><sequence>
       <reply partnerLink="customer"
         portType="lns:loanServicePT"
         operation="request"
         variable="approval">
         <targets>
            <target linkName="setMessage-to-reply" />
            <target linkName="approval-to-reply" />
         </targets>
      </reply>
      </sequence></scope>
   </onAlarm>
</pick>
```

4.2 Mandatory Additional Work—Extension Activity

BPEL leaks a good modeling of the extension activity: the standard elements and standard attributes are included in the extensions and not in BPEL's extensionActivity element. This causes additional work which is explained in the following. This additional work results in a pre- or postprocessing before translating BPEL to BPELscript or vice versa.

4.2.1 Pre- and Postprocessing

Modeling extension activities (discussed in Section 3.4.11) is hard because the *anyElementQName* element may contain BPELs standard attributes and standard elements. This causes two main issues:

i handle XML in a block structured language

ii handle standard attributes

The first is to handle the incoming XML in a block structured manner. Therefore, ANTLR provides a wildcard option shown in Listing 4.5. Thus, everything surrounded by curly brackets will be handled as single object by the lexer. The second problem is to handle the standard attributes and elements inside of the *anyElementsQName*. Thereby using the wildcard makes it hard to get them out of the chunk. Thus, a pre- and postprocessing is required to move the standard elements and attributes out of or in the *anyElementsQName* respectively.

Listing 4.5 ANTLR Wildcarding

```
EXT_ACT
    : '{{{' (options {greedy=false;} : .)* '}}}';
```

Listing 4.6 *a* shows the BPEL specification of the extension activity. For a translation from BPEL to BPELscript the standard attributes and elements have to be moved out of the *anyElementQName* and the *anyElementQName* has to be marked as character data, so that ANTXR can shift it directly into BPELscripts extension activity. This preprocessing is shown in Listing 4.6 *b*. When translating BPELscript to BPEL, the standard attributes and elements cause a similar postprocessing. Starting with the situation in Listing 4.6 *c*, *bosto* will generate BPEL code like in part *b* of the same listing.

Such a pre- and postprocessing can be achieved by XSLT (Extensible Stylesheet Language Transformations [XSL]). XSLT is an XML-based language for XML transformations. However, XSLT processing is not included in *bosto* yet. Note that the presented solution is an interim solution which should be handled by ANTLR's island grammars in the future.

Listing 4.6 Translation of Extension Activity

```
# a - BPEL
  <extensionActivity>
     <anyElementQName standard-attributes>
        standard-elements
     </anyElementQName>
  </extensionActivity>

# b - preprocessed BPEL
  <extensionActivity standard-attributes>
    standard-elements
    <![CDATA[
      <anyElementQName/>
    ]]>
  </extensionActivity>

# c - resulting BPELscript
  std_attr
  {{{
    <anyElementQName/>
  }}};
```

4.3 Optional Additional Work

This section summarizes several suggestions for additional work mentioned during this work.

4.3.1 Abstract Processes

Future work should check if the usage of BPELs abstract processes in BPELscript makes sense and clarify possibilities of an adoption of BPEL extensions to the *BPELscript* approach.

Current proposals are to allow an *abstract* keyword in front of a process to indicate different semantics. The problem with this is, that one has to cope with the fact that abstract processes not describes interfaces, but business logic. This will result in a broad grammar change, since a translator has to cope with different cardinalities for the same rule.

For example, take the pick rule denoted earlier. Listing 4.7 *a* shows the current state of it. To provide a handling of an abstract pick, there are three solutions:

i duplicate rules

ii change cardinalities

iii separate grammar

The first solution to handle abstract processes is to provide each rule twice. On the one hand for executable processes and on the other hand for abstract ones. The rules for executable processes are already provided by the current version of the BPELscript grammar. For abstract rules, the abstract

process root rule should refer to the abstract children rules. An example for the running *pick* example is shown in Listing 4.7 *c*.

The second solution (a) changes the cardinalities to abstract cardinalities and (b) check their correctness by using ANTLRs semantic predicates. The changes are shown in Listing 4.7 *b*. In this case only the cardinality of the *onMessage* statement changes, since the attributes are already optional [5]. Additionally, this cardinalities have to be validated in the AST. Therefore, ANTLR provides *semantic predicates* on the one hand and *actions* on the other hand. Semantic predicates are boolean expressions, evaluated at parse time, that basically turn alternatives on and off [ANT]. Within this handling, the current usage should be validated.

The third solution is derived from solution (i) and uses the concept of "separation of concerns" [HL95]. Thereby it provides a separate grammar to parse interfaces. As a result, the rule a duplicated in two different grammars. One to parse executable processes and the other to parse abstract processes. Note, that only the parser grammar has to be duplicated.

The tree walker and the template file have to be adjusted too, since (a) the cardinalities are derived directly from the parser grammar and (b) the template may provide different behavior for non-empty (+) and empty (* or ?) rules. Listing 4.7 *d* presents the equivalent changes in the tree grammar. In case of the template, there are no changes necessary since *onMessage* do not use non-empty behavior. This results from the MVA processing (see Section 2.4.1) shown in Listing 4.7 *e*.

Listing 4.7 Proposal: Abstract Process Handling

```
# a - the pick rule as it is
  pick :
    CREATE_INST? std_attr
    'pick' '{' onMessage+ onAlarm* '}'

# b - duplicate rules
  abstract_pick :
    'pick' '{' onMessage* onAlarm* '}'

# c - change cardinalities
  pick :
    CREATE_INST? std_attr
    'pick' '{' onMessage* onAlarm* '}'

# d - tree grammar change
  pick :
    ^(PICK onMessage* to+=onAlarm* CREATE_INST? std_attr)

# e - MVA processing of onMessage
  <{<oms:{mes| <mes>}; separator="\n">
```

[5] Note that *std_attr* is not marked as optional here, but in its own rule declaration.

4.3.2 Order of Annotations and in the Header

Currently the order of annotations is fixed to the one given in the BPEL specification. Future work should provide a possibility where the order does not care. Therefore it is recommended to do this by changes in the grammar as follows:

- Allow multiple usages of annotations by using the star operator
- Avoid faulty (multiple) usages of the same annotation by further analysis phases

For example, take the optional attributes *CREATE_INST* and *std_attr* from Listing 4.7. To remove the occurrence-dependency, they have to changed as shown in Listing 4.8. Thereby, additional *actions* or *syntactic predicates* have to check the valid usage.

Listing 4.8 Proposal: Occurrence-Dependencies

```
pick :
  (CREATE_INST std_attr)*
  'pick' '{' onMessage+ onAlarm* '}'
```

Furthermore, the order of *namespaces*, *extensions* and *imports* discussed in Section 3.6.2 should be done in the same manner.

4.3.3 From- and To-Parts of Receive, Reply, Invoke

Section 3.4.5 introduced the usage of the a single variable attribute and *fromPart* and *toPart* elements optionally. BPEL restricts the usage of both is exclusively which is currently not supported in *bosto*. Future work should provide this exclusively usage also in *bosto*.

4.3.4 Correlation—Initiate Pattern

Section 3.7.2 discussed problems when using the proposed shortcut with the XML processing. Future work should find a way to omit the @ tag.

4.3.5 Simulation of Statements and Scopes

Scope simulation was mentioned in conjunction with the *invoke* activity or with scopes including only a single activity. Future work can provide scope simulations by using further analysis phases doing optimization work.

4.3.6 Single Line Structured Statement

The decision to use an implicit sequence (cf. 3.5.1) introduces a blemish which is shown in Listing 4.9. The problem is that a translator will generate a sequence activity for each implicit sequence such as shown in Listing 4.9 *b*. This can be solved in two ways:

- check the number of children
- provide a grammar shortcut

For the first approach one can (a) delegate the work to an optimization phase or (b) embed actions in a translation. As a result the additional sequence can be omitted in cases when only one activity is contained. Thereby, the delegation should be preferred for maintainability reasons, since embedding arbitrary actions will raise the complexity of the tree walker. For the second approach one can provide a Java-like shortcut. The shortcut allows a single statement following direct after the block structured statement without using curly brackets such as shown in Listing 4.9 *c*. From a software engineering point of view this lowers the understandability and introduces a point of failure. When extending a statement such as shown in Listing 4.9 *c* careless without using curly brackets, this will result in wrong code. Therefore, it is recommended to use an optimization phase for that purpose.

Listing 4.9 Problem of Handling Structured Activities with Implicit Sequence

```
# a - Single If Child
  if (...) {a;}

# b - equivalent BPEL
  <if ...><sequence><a/></sequence></if>

# c - shortcut notation
  if (...) a;

# b - equivalent BPEL
  <if ...><a/></if>
```

This chapter gave an overview of the technical realization of *bosto*. The next chapter provides a summary of this work and gives an outlook on future work.

Chapter 5

Summary and Outlook

Summary

The Business Process Execution Language BPEL provides a standardized way for programming in the large in a service oriented world. Since BPEL is an XML-based language, it leaks an adequate modeling without graphical editors. Especially for prototyping or teaching it would be nice to have a programming language which omits the XML-overhead of BPEL but offers the same features as BPEL. Therefore, the *BPEL Simplified Syntax* called SimPEL was recommended. However, SimPEL is not equivalent to BPEL and its aims of specifying business processes. In order to come up with an easy scripting syntax, this work introduced *BPELscript*. *BPELscript* forks directly from SimPEL aiming on big closeness to BPEL. In contrast to SimPEL, *BPELscript* supports all of BPELs constructs including the correlation.

To achieve a 1 : 1 mapping, this work provides a prototypical translator named *bosto*—the BPEL to BPELscript Translator, which uses automated tools as much as possible to achieve in extensibility, maintainability and usability. The second chapter introduced computer translation basics and the main tools. After the basics and tools were presented, the third chapter explains the statements of *BPELscript* with examples and clarify how they can be mapped to BPEL activities, including a discussion of modeling alternatives where necessary.

After the semantic aspects, the fourth chapter discusses the technical description of the translator. Therefore it introduces a theoretical solution how it is realized using parser generators and templates. To show the correctness of the translator and to increase maintainability, that chapter also explains the test concept of *bosto* using grammar unit testing.

Finally, the appendix shows all language elements of BPELscript and the complete BPELscript grammar. Thereby, the grammar presents an overview of the correlation between SimPEL and BPELscript. Furthermore, it provides several overviews about expressions, annotations and keywords. For completeness, the appendix shows a complete *Loan Approval Process* in BPELscript.

Outlook

> If you wait for a complete and perfect concept to germinate in your mind, you are likely to wait forever.
>
> *(T. DeMarco)*

In the real world, there is a gap between abstract design and technical implementation and that the used tools are still in evolution. Since *bosto* itself makes no exception to that tool evolution, there is still work to do in the future.

Variables BPELscript provides implicit variable declaration, but has no type system behind which infers variable types (cf. Sections 3.3.1 and 3.8). Such a system should provide an adequate handling of *portTypes* and WSDL-Files and an interior variable validation and type checking. Thereby, type annotations can be removed if they are inserted during a translation. Also it is recommended to support the *validation* statement since it signals validation for the workflow machine, not for a translator.

extensionExpression *extensionExpressions* are not parsed yet. To improve their usage and to support an optimizing translation in the future, there is to add the capability to parse *extensionExpressions*. Therefore, ANTLR provides the possibility to embed *Island Grammars* which are embedded parsers in the enclosing grammar. For *BPELscript* it seems to be advantageous to add island grammars for Java Script, E4X (an extension to Java Script, refer to [ECM] for more details), XML and XPath. Also a look to the SimPEL grammar is recommended, since it already provides *Island Grammars* for XML and E4X which is currently not provided in BPELscript.

Abstract Processes Section 4.3.1 proposed two solutions. The first solution results in a more complicated grammar, since rules has to be duplicated. The second solution results in an runtime overhead, since statements has to be checked. Future work should check if the usage of BPELs abstract processes in BPELscript makes sense and clarify possibilities of an adoption of BPEL extensions to the *BPELscript* approach.

Bibliography

[ANT] *ANTLR Parser Generator—Website*. http://www.antlr.org/ (Cited on pages 23, 24, 25 und 79)

[Bal05] BALZERT, H.: *Lehrbuch der Objektmodellierung. Analyse und Entwurf*. Heidelberg : Spektrum, 2005 (Cited on page 64)

[BAR] BOISVERT, A. ; ARKIN, A. ; RIOU, M.: *Apache ODE (Orchestration Director Engine)—Website— BPEL Simplified Syntax (simPEL)*. http://ode.apache.org/bpel-simplified-syntax-simbpel.html (Cited on pages 10, 11, 29, 37, 39, 40, 43, 55, 59, 60, 61, 92, 96 und 103)

[BBB+97] BARENDREGT, H. ; BARWISE, J. ; BENTHEM, J. V. ; BLASS, A. ; DANVY, O. ; LESCANNE, P. ; MOOIJ, H. ; MARON, R.t ; PLASMEIJER, R. ; POLLACK, Y.: The impact of the lambda calculus on logic and computer science. In: *Bull. of Symb. Log.* 3 (1997), S. 181–215 (Cited on page 23)

[Bis08] BISCHOF, M.: *Interprozedurale SSA-Form—Das Konzept der statischen Einmalzuweisung*. Hauptseminararbeit, Institut für Softwaretechnologie, Universität Stuttgart, 2008 http://www.iste.uni-stuttgart.de/ps/Lehre/SS2008/HS_Programmanalysen/issa.bischof.pdf (Cited on page 70)

[BPEa] *BPEL, business process management, SOA and you—Website*. http://www.theregister.co.uk/2007/06/13/bpel_bpm_and_you/ (Cited on page 9)

[BPEb] *BPEL: scripting and human tasks—Website*. http://www.theregister.co.uk/2007/06/27/bpel_part_2/ (Cited on pages 10 und 103)

[CFR+91] CYTRON, R. ; FERRANTE, J. ; ROSEN, B. K. ; WEGMAN, M. N. ; ZADECK, F. K.: Efficiently computing static single assignment form and the control dependence graph. In: *ACM Trans. Program. Lang. Syst.* 13 (1991), Nr. 4, S. 451–490. http://dx.doi.org/http://doi.acm.org/10.1145/115372.115320. – DOI http://doi.acm.org/10.1145/115372.115320. – ISSN 0164–0925 (Cited on page 70)

[CKLW03] CURBERA, F. ; KHALAF, R. ; LEYMANN, F. ; WEERAWARANA, S.: Exception Handling in the BPEL4WS Language. In: *International Conference on Business Process Management* Bd. 2678, 2003 (LNCS), S. 276–290 (Cited on page 39)

[Dij68] DIJKSTRA, Edsger W.: Letters to the editor: go to statement considered harmful. In: *Commun. ACM* 11 (1968), Nr. 3, 147–148. http://doi.acm.org/10.1145/362929.362947. – ISSN 0001–0782 (Cited on page 40)

[DOM] *Document Object Model (DOM)*. http://www.w3.org/DOM/ (Cited on page 20)

[ECM] *Standard ECMA-357—ECMAScript for XML (E4X) Specification*. http://www.ecma-international.org/cgi-bin/counters/unicounter.pl?name=Ecma-357&deliver=http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-357.pdf (Cited on pages 34 und 84)

[Fow99] FOWLER, M.: A UML testing framework. In: *Softw. Dev.* 7 (1999), Nr. 4, S. 41–46. – ISSN 1070–8588. – Miller Freeman, Inc. (Cited on page 26)

[GR93] GRAY, J. ; REUTER, A.: *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993 (Cited on page 60)

[gUn] *gUnit—Grammar Unit Testing*. http://www.antlr.org/wiki/display/ANTLR3/gUnit+-+Grammar+Unit+Testing (Cited on pages 13, 25, 26 und 104)

[Hid07] *The Hidiocy of XML Languages*. http://blog.objectmentor.com/articles/2007/05/17/the-hidiocy-of-xml-languages. Version: 2007 (Cited on page 29)

[HL95] HURSCH, W. L. ; LOPES, C. V.: Separation of concerns. (1995) (Cited on page 79)

[Jav] *The Java Language Specification, Third Edition*. http://java.sun.com/docs/books/jls/third_edition/html/interfaces.html (Cited on page 38)

[JJ75] JOHNSON, S. C. ; JOHNSON, S. C.: Yacc: Yet Another Compiler-Compiler. (1975) (Cited on page 15)

[KKL06] KHALAF, R. ; KELLER, A. ; LEYMANN, F.: Business processes for web services: principles and applications. In: *IBM Syst. J.* 45 (2006), Nr. 2, S. 425–446. – ISSN 0018–8670 (Cited on page 9)

[KMWL08] KOPP, O. ; MARTIN, D. ; WUTKE, D. ; LEYMANN, F.: *On the Choice Between Graph-Based and Block-Structured Business Process Modeling Languages*. Modellierung betrieblicher Informationssysteme (MobIS 2008), 2008 (Cited on pages 29 und 39)

[Lam83] LAMPSON, B. W.: Hints for computer system design. In: *IEEE Software*, 1983, S. 33–48 (Cited on page 69)

[Lea] LEA, D.: A Java Fork/Join Framework. http://gee.cs.oswego.edu/dl/papers/fj.pdf. – State University of New York (Cited on page 39)

[Ley01] LEYMANN, F.: Web Services Flow Language (WSFL 1.0). (2001), May. http://www.ibm.com/software/solutions/webservices/pdf/WSFL.pdf. – IBM Software Group (Cited on page 9)

[Ley06] LEYMANN, F.: Choreography for the Grid: towards fitting BPEL to the resource framework: Research Articles. In: *Concurr. Comput. : Pract. Exper.* 18 (2006), Nr. 10, 1201–1217. http://dx.doi.org/10.1002/cpe.v18:10. – ISSN 1532–0626 (Cited on page 9)

[LK] LESSEN, T. van ; KOPP, O.: *Apache ODE (Orchestration Director Engine)—Website— BPEL Simplified Syntax (simBPEL)*. http://ode.apache.org/

bpel-simplified-syntax-simbpel.html (Cited on pages 10, 11, 29, 31, 39, 43, 51, 92, 96 und 103)

[LR00] LEYMANN, F. ; ROLLER, D.: *Production workflow: concepts and techniques*. Upper Saddle River, NJ, USA : Prentice Hall PTR, 2000 (Cited on pages 39, 60 und 63)

[Mag00] MAGLOIRE, A.: *Grep: Searching for a Pattern*. Iuniverse Inc, 2000 (Cited on page 69)

[McC60] MCCARTHY, J.: Recursive functions of symbolic expressions and their computation by machine, Part I. In: *Commun. ACM* 3 (1960), Nr. 4, 184–195. http://dx.doi.org/http://doi.acm.org/10.1145/367177.367199. – DOI http://doi.acm.org/10.1145/367177.367199. – ISSN 0001–0782 (Cited on page 23)

[Muc97] MUCHNICK, S. S.: *Advanced compiler design and implementation*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1997 (Cited on page 13)

[Nag03] NAGL, M.: *Softwaretechnik mit ADA 95*. Vieweg, 2003 (Cited on page 44)

[NNC05] NIELSON, F. ; NIELSON, H. R. ; C.HANKIN: *Principles of Program Analysis*. 2. Springer, 2005 (Cited on page 64)

[OAS07a] OASIS WEB SERVICES BUSINESS PROCESS EXECUTION LANGUAGE (WSBPEL) TC: Web Services Business Process Execution Language Version 2.0—OASIS Standard. (2007). http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html (Cited on pages 9, 29, 31, 33, 37, 38, 39, 43, 44, 45, 47, 48, 49, 52, 53, 57, 59, 60, 62, 63, 64, 65, 67 und 104)

[OAS07b] OASIS WEB SERVICES BUSINESS PROCESS EXECUTION LANGUAGE (WSBPEL) TC: Web Services Business Process Execution Language Version 2.0—Primer. (2007). http://docs.oasis-open.org/wsbpel/2.0/Primer/wsbpel-v2.0-Primer.pdf (Cited on pages 31, 32, 43, 44, 45, 47, 51, 55, 56, 57, 59, 60, 62, 63 und 67)

[Obj08] OBJECT MANAGEMENT GROUP (Hrsg.): *Business Process Modeling Notation, V1.1 – OMG Available Specification*. : Object Management Group, Januar 2008 (Cited on page 9)

[ODBt06] OUYANG, C. ; DUMAS, M. ; BREUTEL, S. ; TER HOFSTEDE, A.: Translating Standard Process Models to BPEL. In: *Proceedings 18th International Conference on Advanced Information Systems Engineering (CAiSE)* Bd. 4001, Springer Verlag, June 2006 (Lecture Notes in Computer Science) (Cited on page 9)

[ODEa] *Apache ODE (Orchestration Director Engine)—Auction.simpel*. http://svn.apache.org/repos/asf/ode/sandbox/simpel/src/test/resources/auction.simpel (Cited on pages 66 und 105)

[ODEb] *Apache ODE (Orchestration Director Engine)—Website*. http://ode.apache.org/ (Cited on pages 10, 29 und 96)

[Par04] PARR, T.: Enforcing Strict Model-View Separation in Template Engines. (2004). http://www.cs.usfca.edu/~parrt/papers/mvc.templates.pdf. – University of San Francisco (Cited on page 23)

[Par06] PARR, T.: A Functional Language For Generating Structured Text. (2006). http://www.cs.usfca.edu/~parrt/papers/ST.pdf. – University of San Francisco (Cited on pages 13 und 22)

[Par07] PARR, T.: *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Raleigh : The Pragmatic Bookshelf, 2007 http://www.pragprog.com/titles/tpantlr/the-definitive-antlr-reference (Cited on pages 11, 13, 14, 15, 16, 17, 18 und 103)

[Par08] PARR, T.: The Reuse of Grammars with Embedded Semantic Actions. (2008). http://www.cs.vu.nl/icpc2008/docs/Parr.pdf. – University of San Francisco (Cited on page 17)

[PB07] PARR, T.e ; BOVET, J.: ANTLRWorks: An ANTLR Grammar Development Environment (UNPUBLISHED DRAFT). (2007). http://www.antlr.org/papers/antlrworks-draft.pdf (Cited on page 15)

[PZ00] PRATT, T. ; ZELKOWITZ, M.: *Programming Languages: Design and Implementation*. 4. Prentice Hall, 2000

[RE] RIOU, M. ; ET AL.: *Apache ODE (Orchestration Director Engine)—Website—SimPEL Grammar*. http://svn.apache.org/viewvc/ode/sandbox/simpel/src/main/antlr/org/apache/ode/simpel/antlr/SimPEL.g?revision=707517 (Cited on pages 11, 29, 91, 92, 96, 103 und 105)

[Rio08] RIOU, M.: *Re: Correlation-Set/Mapping in SimPEL grammar*. Apache ODE (Orchestration Director Engine)—Mailing List, 2008 http://mail-archives.apache.org/mod_mbox/ode-dev/200806.mbox/<fbdc6a970806091549q20917af9q58843ddc5a1e203a@mail.gmail.com> (Cited on pages 29 und 64)

[SAX] *Simple API for XML (SAX)*. http://www.saxproject.org/ (Cited on page 20)

[Sch08] SCHUMM, D.: Graphische Modellierung von BPEL Prozessen unter Verwendung der BPMN Notation. (2008). ftp://ftp.informatik.uni-stuttgart.de/pub/library/medoc.ustuttgart_fi/DIP-2720/DIP-2720.pdf. – Diplomarbeit, Institut für Architektur von Anwendungssystemen, Universität Stuttgart (Cited on page 9)

[SE] STANCHFIELD, S. ; ET AL.: *ANTXR: Easy XML Parsing, based on the ANTLR parser generator—Website*. http://www.antlr.org/ (Cited on pages 13, 19, 20, 21, 22, 70, 71 und 104)

[SGSE07] S., Staiger ; G., Vogel ; S., Keul ; E., Wiebe: Interprocedural Static Single Assignment Form. In: *Proceedings of the 14th Working Conference on Reverse Engineering*, 2007, 1–10 (Cited on page 70)

[SLAU06] SETHI, R. ; LAM, M. ; AHO, A. ; ULLMANN, J.: *Compilers: Principles, Techniques, and Tools*. 2. Pearson Studium, 2006 (Cited on page 13)

[Str] *String Template Engine—Website*. http://www.stringtemplate.org/ (Cited on page 22)

[Tha01] THATTE, S.: XLANG Web Services for Business Process Design. (2001). – Microsoft Corporation (Cited on page 9)

[Via08] VIAL, T.: *gUnit freezing when choking on invalid input*. ANTLR-Interest Mailing List, 2008 http://www.antlr.org:8080/pipermail/antlr-interest/2008-June/028858.html (Cited on page 26)

[WCL+05] WEERAWARANA, S. ; CURBERA, F. ; LEYMANN, F. ; STOREY, T. ; FERGUSON, D. F.: *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More*. Upper Saddle River, NJ, USA : Prentice Hall PTR, 2005 (Cited on page 9)

[Whi] WHITE, S. A.: Using BPMN to Model a BPEL Process. `http://www.bpmn.org/Documents/MappingBPMNtoBPELExample.pdf`. – IBM Corp. (Cited on page 9)

[WM96] WILHELM, R. ; MAURER, D.: *Übersetzerbau. Theorie, Konstruktion, Generierung*. Springer, 1996 (Cited on page 13)

[WM08] WHITE, S. A. ; MIERS, D.: *BPMN Modeling and Reference Guide Understanding and Using BPMN*. Future Strategies Inc., Lighthouse Pt, FL, 2008 (Cited on page 9)

[XPAa] *Xml Processing for Antlr—Website*. `http://xpa.sourceforge.net/` (Cited on pages 19 und 20)

[XPab] *XML Path Language (XPath) Version 1.0*. `http://www.w3.org/TR/xpath` (Cited on page 34)

[XSL] *XSL Transformations (XSLT) Version 2.0*. `http://www.w3.org/TR/xslt20/` (Cited on page 77)

All links were last followed on October 14, 2008.

Appendix A

Appendix

A.1 Language Basics

A.1.1 Identifier

Listing A.1 Identifier in BPELscript [RE]

```
ID : (LETTER | '_' ) (LETTER | DIGIT | '_' | '-' )*;
```

A.1.2 Strings and Escape Sequence

Listing A.2 Strings in BPELscript [RE]

```
STRING : '"' ( ESCAPE_SEQ | ~('\\'|'"') )* '"';

ESCAPE_SEQ : '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\');
```

A.1.3 Comments

Listing A.3 Single Line Comments in BPELscript [RE]

```
SL_COMMENTS : ('#'|'//') .* CR { $channel = HIDDEN; };

CR : ('\r' | '\n' )+ { $channel = HIDDEN; };
```

A.2 List of Expressions

Conditional Expr	Arithmetic Expr
==	+
!=	−
<	*
>	/
<=	
>=	

Table A.1: Conditional and Arithmetic Expressions [RE]

Name	Syntax	Section
Extension Expression	[...]	3.3.2
Extension Activity	{{{...}}}	3.4.11

Table A.2: Extension Expression and Activity [BAR, LK]

A.3 List of Annotations

Annotation	Argument	Type	Shortcut	Section
faultVariable	@faultVariable	ID	@faultVar	3.4.8, 3.6.3
queryLanguage	@queryLanguage	STRING		3.4.1
expressionLanguage	@expressionLanguage	STRING		3.4.1
faultMessageType	@faultMessageType	STRING		3.6.3
faultElement	@faultElement	STRING		3.6.3
messageType	@messageType	STRING		3.3.1, 3.6.5
element	@element	STRING		3.6.5
type	@type	STRING		3.3.1, 3.8
portType	@portType	STRING	@pt	3.4.5, 3.6.5
messageExchange	@messageExchange	STRING	@mex	3.4.5, 3.6.1
faultName	@faultName	STRING	@fault	3.4.5, 3.4.8, 3.6.3
standard attributes				3.4.1
name	@name	STRING		
suppressJoinFailure	@suppressJoinFailure		@dpe	
exitOnStandardFault	@exitOnStandardFault		@exit	3.6
mustUnderstand	@mustUnderstand			
parallel	@parallel		@par	3.5.6
successfulBranchesOnly	@successfulBranchesOnly		@sbo	3.5.6
initializePartner	@initializePartner		@init	3.2
initiate				3.7.2
force	force		!	
join	join		?	
pattern				3.7.2
request	request		@>	
response	response		@<	
request-response	request-response		@><	
isolated	@isolated			3.6
createInstance	@createInstance		@ci	3.4.5, 3.5.4
validate	@validate			3.3.2, 3.4.7
keepSourceElementName	@keepSrcElementName		@keepSrc	3.3.2
ignoreMissingFromData	@ignoreMisssingFromData		@ignore	3.3.2
property	@property			3.3.2

Table A.3: List of Annotations

A.4 List of Keywords

Keyword	Section	Keyword	Section	Keyword	Section
alarm	3.4.9	for	3.5.6	receive	3.4.5
and	3.5.5	if	3.5.2	repeatEvery	3.4.9
catch	3.6.3	in	3.4.5	reply	3.4.5
compensate	3.6.2	inout	3.4.5	scope	3.6
compensation	3.6.2	import	3.8	signal	3.4.2
correlates	3.7.1	invoke	3.4.5	throw	3.4.8
do	3.5.3	join	3.4.2	timeout	3.4.9
else	3.5.2	messages	3.6.1	to	3.5.6
elseif	3.5.2	namespace		try	3.6.3
event	3.6.5	nop	3.4.10	until	3.5.3
events	3.6.5	out	3.4.5	validate	3.4.7
exit	3.4.12	parallel	3.5.5	var	3.3.1
extension		partnerlink	3.2	while	3.5.3
finally	3.6.4	pick	3.5.4	with	3.4.5
finish	3.5.6	process	3.1		

Table A.4: List of Keywords

A.5 Loan Approval Process in BPELscript

Listing A.4 Example BPELscript—Complete Loan Approval Process

```
namespace pns = "http://example.com/loan-approval/";
namespace lns = "http://example.com/loan-approval/wsdl/";

@type "http://schemas.xmlsoap.org/wsdl/"
import lServicePT = lns::"loanServicePT.wsdl";

@suppressJoinFailure
process pns::loanApprovalProcess {

partnerLink customer = (lns::loanPartnerLT, loanService , null),
            approver = (lns::loanApprovalLT, null, approver),
            assessor = (lns::riskAssessmentLT, null, assessor);
try {
    parallel {
      @portType "lns:leanServicePT" @createInstance
      request = receive(customer, request);
      signal(receive-to-assess, [$request.amount &lt; 10000]);
      signal(receive-to-approval, [$request.amount >= 10000]);
    } and {
      join(receive-to-assess);
      @portType "lns:riskAssessmentPT"
      risk = invoke(assessor, check, request);
      signal(assess-to-setMessage, [$risk.level = 'low']);
      signal(assess-to-approval, [$risk.level != 'low']);
    } and {
      join(assess-to-setMessage);
      approval.accept = "yes";
      signal(setMessage-to-reply);
    } and {
      join(receive-to-approval, assess-to-approval);
      @portType "lns:loanApprovalPT"
      approval = invoke(approver, approve, request);
      signal(approval-to-reply);
    } and {
      join(approval-to-reply, setMessage-to-reply);
      @portType "lns:loanServicePT"
      reply(customer, request, approval);
    }
  } @faultMessageType "lns:errorMessage"
    catch(lns::loanProcessFault) { |error|
    @portType "lns:loanServicePT" @fault "unableToHandleRequest"
    reply(customer, request, error);
  }
}
```

A.6 BPELscript Grammar

BPELscript results mainly from the enhancement of the SimPEL grammar [RE] and the proposals of the ODE [ODEb, BAR, LK]. Therefore, the following shows the BPELscript grammar enlarged by special symbols indicating the correlation. This symbols are *not* part of the ANTLR grammar.

- grammar rules containing a # following the rule name, are completely adopted from SimPEL
- grammar rules containing a / following the rule name, are inspired by SimPEL. Inside of those rules, parts resulting from SimPEL are marked by an #.
- grammar rules containing a $ following the rule name, are not contained in SimPEL

```
grammar BPELscript;

// MAIN BPEL SYNTAX
program #
    :   declaration+;
declaration /
    :   process#
    |   sub_declaration;

sub_declaration $
    :   namespace | extension | imports;

// Process
process /
    :   ('@queryLanguage' queryLg=STRING)?
        ('@expressionLanguage' exprLg=STRING)?
        sjf=SJF?
        exitOnStandardFault=EOSF?
        'process' ns_id #
        std_attr j+=ajoin? s+=asignal*
        block #
        eventHdl?;

proc_stmts $
    :   (join SEMI)? proc_stmt (s+=signal SEMI)*;

proc_stmt #
    :   //structured stmts
        if_ex | flow | pick | while_ex | until_ex | foreach #
        | scope_ex | try_ex #
        | ext_act | corr_sets
        //simple stmts
        | ((invoke | receive | reply | assign | throw_ex | rethrow_ex #
        | exit | variables | partner_links #
        | alarm | timeout | validate | compensate | nop | messages)
        SEMI!); #

block #
    :   '{' proc_stmts+ '}';

scope_block $
```

```
    :   '{' sub_declaration* proc_stmts+ '}';

param_block #
    :   '{' ('|' in+=ID (',' in+=ID)* '|')? proc_stmts+ '}';

body #
    :   block | proc_stmts;

// Structured activities
pick /
    :   CREATE_INST? std_attr
        'pick' '{' onMessage+ onAlarm* '}'; #


onMessage /
    :   portType? msgEx?
        'onMessage' '(' p=ID ',' o=ID (',' correlation)? ')' with_ex? param_block; #

onAlarm /
    :   // at least one expression must be there
        (alarm | timeout)? repeatEvery? scope_short;

alarm /
    :   std_attr
        'alarm' '(' expr ')'; #

timeout /
    :   std_attr
        'timeout' '('expr ')'; #

repeatEvery $
    :   'repeatEvery' '(' expr ')';

flow /
    :   std_attr
        'parallel' s+=sequence ( 'and' s+=sequence)*; #

signal #
    :   'signal' '('ID (',' expr)? ')';

asignal /
    :   '@signal' '('ID (',' expr)? ')';

ajoin /
    :   '@join' '(' k+=ID (',' k+=ID)* (',' expr)? ')';

join #
    :   'join' '(' k+=ID (',' k+=ID)* (',' expr)? ')';

if_ex /
    :   std_attr
        'if' '(' iex=expr ')' s=sequence #
        ('elseif' '(' eiex+=expr ')' sei+=sequence)*
        ('else' se=sequence)?; #
```

```
sequence $
    :   std_attr j+=ajoin? s+=asignal*
        b=body;

scope_sequence $
    :   j+=ajoin? s+=asignal*
        b=scope_block;

while_ex /
    :   std_attr
        'while' '(' expr ')' s=sequence; #



until_ex /
    :   std_attr
        'repeat' s=sequence 'until' '(' expr ')'; #

foreach /
    :   PARALLEL?
        successfulBranchesOnly=SBO?
        std_attr
        'for' '(' cName=ID '=' init=expr ('to'|SEMI) #
        cond=expr (('finish'|SEMI) complete+=expr)? ')' scope_short; #

try_ex /
    :   'try' body catch_ex* catchAll?; #

catch_ex /
    :   (('@faultMessageType' fMT=STRING) | faultElt)?
        'catch' '(' ns_id ')' param_block; #

catchAll $
    :   'catchAll' block;

scope_ex /
    :   ISOLATED? EOSF? SJF?
        'scope' ('(' ID? ')')? scope_sequence scope_stmt; #

scope_short $
    :   scope_sequence scope_stmt;

scope_stmt /
    :   compensation? termination? eventHdl?;

termination $
    :   'onTermination' body;

eventHdl $
    :   'events' '{' onEvent* onAlarm* '}';

onEvent /
    :   portType? msgEx?
        ( msgType | viElt )?
        (var=ID '=' )? 'event' '(' p=ID ',' o=ID (',' correlation)? ')' with_ex? scope_short; #
```

```
compensation # //compensation handler
    :   'compensation' body;

with_ex /
    :   'with' '(' wm+=with_map (',' wm+=with_map)* ')'; #

with_map /
    :   ID ':' KEY? path_expr; #



// Simple activities
receive /
    :   portType? CREATE_INST? msgEx? std_attr
        'receive' '(' p=ID ',' o=ID (',' correlation)? ')' with_ex?; #

reply /
    :   portType? faultName? msgEx? std_attr
        'reply' '(' p=ID ',' o=ID (',' in=ID)? (',' correlation)? ')'  with_ex?; #

invoke /
    :   portType? std_attr
        'invoke' '(' p=ID ',' o=ID (',' in=ID)? (',' correlation)? ')' with_ex? compensation?; #

assign /
    :   portType? CREATE_INST? VALID? KEEPSRC? IGNORE? faultName?
        msgEx? std_attr //only receive and invoke
        path_expr PART? '=' rvalue; #

rvalue /
    :   receive #
    |   invoke #
    |   expr PART?; #

throw_ex /
    :   (('@faultVariable' |'@faultVar') faultVar=ID)? std_attr
       'throw' '(' ns_id ')'; #

rethrow_ex $
    :   std_attr
        'rethrow';

compensate /
    :   std_attr
        'compensate' ('(' target=ID ')')?; #

exit /
    :   std_attr
        'exit'; #

validate $
    :   std_attr
```

```
        'validate' v+=ID (',' v+=ID)*;

ext_act $
    :   std_attr
        e=EXT_ACT;

nop $
    :   std_attr
        'nop';

// Others
namespace #
    :   'namespace' ID '=' STRING SEMI;

extension $
    :   MUSTUND?
        'extension' ID '=' STRING SEMI;

imports $
    :   viType
        'import' (id=ID '=' (ns=ID '::' )? loc=STRING ) SEMI;

messages $ //Exchange
    :   'messages' m+=message (',' m+=message)*;

message $
    :   ID;

variables #
    :   'var' v+=variable (',' v+=variable)*;

variable /
    :   msgType? viType? viElt?
        ID  with_ex?; #

partner_links #
    :   ('partnerLink' | 'partnerlink') pl+=partner_link (',' pl+=partner_link)*;

partner_link $
    :   ID '=' '(' plType=ns_id? (',' roleA=ns_id)? (',' roleB=ns_id)?
        (',' init=INITPARTNER)? ')';

correlation #
    :   '{' corr_mapping (',' corr_mapping)* '}';

corr_mapping /
    :   init=INIT_COR?
        pattern=PATTERN_COR?
        f1=ID; #

corr_sets $
    :   'correlates' '{'cs+=corr_set ';' (cs+=corr_set ';')* '}';

corr_set $
    :   f=ID '(' par+=ID (',' par+=ID)* ')';
```

```
#// Expressions
expr        :       s_expr | EXT_EXPR | funct_call;
funct_call  :       p+=ID '(' p+=ID* ')';
s_expr      :       condExpr;
condExpr    :       aexpr ( ('==' ^|'!=' ^|'<' ^|'>' ^|'<=' ^|'>=' ^) aexpr )?;
aexpr       :       mexpr (('+'|'-') ^ mexpr)*;
mexpr       :       atom (('*'|'/') ^ atom)* | STRING;
atom        :       path_expr | INT | '(' s_expr ')';
path_expr   :       pelmt+=ns_id ('.' pelmt+=ns_id)*;
ns_id       :       (pr=ID '::')? loc=ID;


$// optional attributes
portType    :       ('@portType' | '@pt') STRING;
std_attr    :       ('@name' name=STRING)? suppressJoinFailure=SJF?;
msgEx       :       ('@messageExchange' | '@mex') STRING;
msgType     :       ('@messageType' | '@msgType') msgT=STRING;


$// var or import type
viType      :       '@type' type=STRING;
viElt       :       '@element' elt=STRING;
faultName   :       ('@faultName' | '@fault') STRING;
faultElt    :       '@faultElement' STRING;

// LEXER RULES
EXT_EXPR#   :        '[' (options {greedy=false;} : .)* ']';
EXT_ACT$    :        pre='{{{' (options {greedy=false;} : c=.)* post='}}}';

#// Basic tokens
KEY$            :        'in' | 'out' | 'inout';
SEMI            :       ';';
ID              :       (LETTER | '_' ) (LETTER | DIGIT | '_' | '-' )*;
INT             :       (DIGIT )+ ;
STRING          :       '"' ( ESCAPE_SEQ | ~('\\'|'"') )* '"';
ESCAPE_SEQ      :       '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\');
SL_COMMENTS     :       ('#'|'//') .* CR { $channel = HIDDEN; };
CR              :       ('\r' | '\n' )+ { $channel = HIDDEN; };
WS              :       ( ' ' | '\t' )+ { skip(); };
fragment DIGIT  :       '0'..'9';
fragment LETTER :       'a'..'z' | 'A'..'Z';

$// Boolean annotations omitted
```

List of Figures

List of Tables

List of Listings

Index

Declaration

All the work contained within this thesis,
except where otherwise acknowledged,
was solely the effort of the author.
At no stage was any collaboration entered into with
any other party.

(Marc Bischof)