BPELscript: A Simplified Script Syntax for WS-BPEL 2.0

Marc Bischof, Oliver Kopp, Tammo van Lessen, Frank Leymann
Institute of Architecture of Application Systems
University of Stuttgart
Universitätsstraße 38, 70569 Stuttgart, Germany
http://www.iaas.uni-stuttgart.de

in: 35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2009).
See also BibTeX entry below.

BPELscript: A Simplified Script Syntax for WS-BPEL 2.0

Marc Bischof, Oliver Kopp, Tammo van Lessen, Frank Leymann

Institute of Architecture of Application Systems
Universität Stuttgart
Stuttgart, Germany
mc.bischof@googlemail.com, {lastname}@iaas.uni-stuttgart.de

***Abstract*—Business processes are usually modeled using graphical notations such as BPMN. As a first step towards execution as workflow, a business process is transformed to an abstract WS-BPEL process. Technical details required for execution are added by an IT expert. While IT experts expect Java-like syntax for programs, WS-BPEL requires processes to be expressed in XML. This paper introduces BPELscript as a new syntax for WS-BPEL aiming to reduce the barrier for IT experts to use WS-BPEL by providing a JavaScript-inspired syntax.**

***Keywords*-service orchestration; service scripting; BPEL; BPM lifecycle**

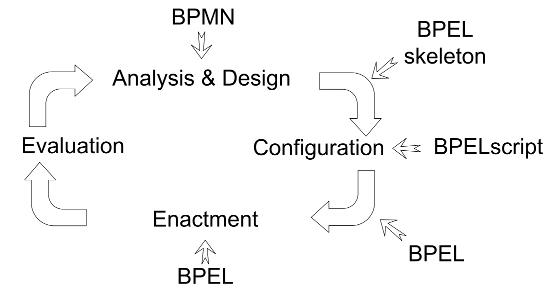

Figure 1. Position of BPELscript in the BPM lifecycle

I. Introduction

The business process management (BPM) lifecycle consists of the phases analysis & design, configuration, enactment and evaluation [1]. One architectural style to support BPM is the Service-oriented Architecture (SOA) and the Web service stack is the most prominent implementation of the SOA [2]. For definition of workflows, the Web service stack provides WS-BPEL (BPEL for short, [3]). The main features of BPEL are its native support for concurrency, backward and forward recovery. In the BPM lifecycle, BPEL is used for workflow enactment. Before a workflow can be enacted, it has to be implemented based on a process model during the configuration phase. To ease the implementation of a business process as a workflow, the process model is transformed to an abstract BPEL process model. After the mapping, an IT expert refines the abstract BPEL process model to an executable BPEL process model containing all technical details required for execution.

While BPEL is the de-facto language for workflow enactment in the area of Web services [2], there are several languages proposed for modeling business processes. The most prominent language is the business process modeling notation BPMN [4]. For that language, a mapping to BPEL is available [5]. A BPEL process is serialized using XML. Thus, an IT expert either has to use a BPEL designer or has to edit pure XML code for refining the resulting BPEL model to an executable BPEL model. Some of the available BPEL modeling tools however do not support all features of BPEL [6]. For example, the BPEL designer of NetBeans[1] does not support defining explicit control links to connect activities.

[1]http://www.netbeans.org/features/soa/

In this paper, we present BPELscript as an alternative to modify BPEL process models. BPELscript offers the same constructs as BPEL, but comes with a syntax inspired by scripting languages such as JavaScript or Ruby. The semantics of BPELscript is given by its mapping to BPEL. Each construct of BPELscript can be mapped to BPEL. Since BPELscript itself covers all features of BPEL, each construct of BPEL can also be mapped to BPELscript. Thus, a mapping in both directions is provided. The targeted embedding in the BPM lifecycle is presented in Figure 1: the business process model is modeled in BPMN and exported to BPEL. The BPEL process model is then used as interchange format for the configuration phase. Here, the BPEL process model is converted to BPELscript. Now, the IT expert can use his favorite IDE to edit the BPELscript file and enriches the process definition by technical details such as defining variables with XML schema data types, binding interactions to concrete WSDL operations and refining the process logic, e.g. by adding more sophisticated data manipulation tasks. After he has finished this refinement—also known as "Executable Completion"—the BPELscript is converted back to a BPEL process model. This BPEL process model is then deployed to a workflow engine. Having a mapping from BPEL to BPELscript and vice versa at hand, BPELscript does not force companies to implement new translations from BPMN to other workflow languages, switch to new workflow engines or change their process monitoring solutions. BPELscript is seamlessly integrated into the established BPM lifecycle.

A sample process is shown using BPMN, BPEL and BPELscript in Section II. There have been other approaches to define a new syntax for BPEL. A summary of these approaches is given in Section III. BPEL offers the

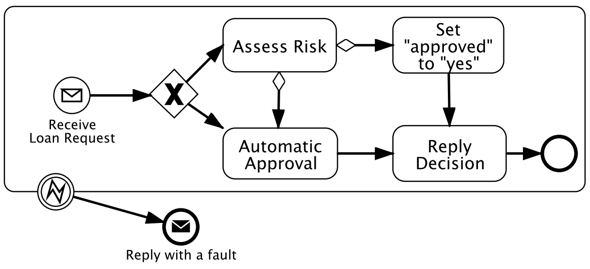

Figure 2. Loan approval process modeled in BPMN

capabilities to define a graph defining a workflow. We provide more details about the graph-based capabilities of BPEL and their representation in BPELscript in Section IV. The representation of other language specifics such as the basic activities and optional attributes are presented in Section V. Subsequently, we provide an overview on the transformations between BPELscript and BPEL in Section VI. Finally, Section VII concludes and provides an outlook on future work.

II. Loan Approval in BPELscript

The loan approval process is a simple business process used in the BPEL specification to give an overview on the capabilities of the language. Figure 2 presents the process modeled using BPMN. First a credit request message is received. In the case of a high amount, the loan request is directly sent to a human approver. If the amount is low, the risk of the loan is assessed. In the case of a low risk, the loan is directly approved. If not, a human approver has to check the loan request. The decision of the human approver is directly sent as reply to the customer. In the case of a fault during the processing, the customer is notified about the fault.

The process expressed in BPEL serialized in XML is 90 lines long (cf. [3]). An excerpt is shown in Figure 3. The attribute `suppressJoinFailure="yes"` enables dead-path elimination required for graph-oriented programming. Details are described in Section IV. The `flow` activity provides support for concurrency and synchronization. Each activity may have incoming and outgoing links. In the process, the `receive` activity is used to receive the loan request. The attribute `operation` denotes which WSDL operation is used to receive the request. `variable` denotes the variable, where the received message should be stored. Finally, `createInstance="yes"` denotes that a new process instance has to be created upon receipt of the message. The receive activity itself has two outgoing links listed in the `sources` element. One for the case that a loan with an amount less than 10,000 € and the other one for an amount equal or greater than 10,000 €. These conditions are stored in the `transitionCondition` element. If the amount is less than 10,000 €, the status of the link `receive-to-asses` is set to true. Since the `invoke` only has one incoming link and no explicit join condition, the `invoke` activity is executed. It calls the operation `check` at the risk checking service. The message to be sent is specified via the `inputVariable` attribute.

```
<process suppressJoinFailure="yes">
  <faultHandlers>
    <catch faultName="lns:loanProcessFault" >
      <reply operation="request" variable="error"
       faultName="unableToHandleRequest" />
    </catch>
  </faultHandlers>
  <flow>
    <links>
      <link name="receive-to-assess"/>
      <link name="receive-to-approval" />
      ...
    </links>
    <receive operation="request" variable="request"
      createInstance="yes">
      <sources>
        <source linkName="receive-to-assess">
          <transitionCondition>
            $request.amount &lt; 10000
          </transitionCondition>
        </source>
        <source linkName="receive-to-approval">
          <transitionCondition>
            $request.amount &gt;= 10000
          </transitionCondition>
        </source>
      </sources>
    </receive>
    <invoke operation="check"
      inputVariable="request"
      outputVariable="risk">
      <targets>
         <target linkName="receive-to-assess"/>
      </targets>
      <sources>
        <source linkName="assess-to-setMessage">
          <transitionCondition>
            $risk.level='low'
          </transitionCondition>
        </source>
        <source linkName="assess-to-approval">
          <transitionCondition>
            $risk.level!='low'
          </transitionCondition>
        </source>
      </sources>
    </invoke>
    ...
  </flow>
</process>
```

Figure 3. Simplified version of the loan approval process as BPEL workflow

The received message is stored in the `outputVarible`. The two outgoing links of the `invoke` activity are again listed in the `sources` element. Figure 4 presents the *full* process in BPELscript syntax. Lines 1 to 5 provide necessary namespace declarations and WSDL imports. `@suppressJoinFailure` enables dead-path elimination, described in Section IV. `parallel` is the BPELscript pendant of the `flow` activity. Incoming links are denoted by the `join` statement. Outgoing links are denoted by `signal` statements. Details are also described in Section IV. The remaining elements of BPEL processes are shown in Section V.

III. Related Work

The only existing language close to BPELscript with a compatibility close to BPEL is SimPEL [7]. However, the

```
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::loanServicePT @createInstance
    request = receive(customer, request);
    signal(receive-to-assess,
      [$request.amount < 10000]);
    signal(receive-to-approval,
      [$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);
  }
}
```

Figure 4. Loan approval in BPELscript

Apache ODE team "preferred language consistency over BPEL compatibility" [7], SimPEL can only be used in Apache ODE and is not fully compatible with BPEL. In contrast to SimPEL, BPELscript a) is fully transformable to BPEL and thus is deployable to any BPEL-compatible workflow engine and b) fully supports abstract processes.

The Simple Service Composition Language (SSCL [8]) is a "simple programming language for Web Service composition", which provides a subset of the Coopetition Language (CL [9]) in a syntax close to Basic. CL in turn provides a subset of BPEL's capabilities [9]: the capabilities of built-in eventhandling, fault handling and compensation handling have been dropped. While BPELscript supports all control flow and data handling constructs of BPEL, SSCL provides "two control-flow constructs: if-then-else [...] and while", but no support for data manipulation. A translation from SSCL to CL is available, whereas a translation of CL to SSCL is not. Thus, SSCL cannot be integrated fully in the BPM lifecycle, whereas BPELscript can.

The BPEL to Java (B2J) subproject[2] offers a translation of executable BPEL to executable Java classes. While the Java classes can be modified, the Java classes cannot be transformed back to a single BPEL process. Thus, the modifications cannot be deployed on workflow engines. In addition, the translation is not capable to deal with abstract BPEL processes.

There exist several formalizations for BPEL. By mapping BPEL to a formal model, BPEL gets a new syntax, too. An overview of existing formalizations is given in [10]. The only translation from a formal syntax back to BPEL is known for Open Workflow Nets [11] and is presented in [12]. Open Workflow Nets (oWFNs) are based on workflow nets [13], which in turn are based on Petri nets. Since the oWFNs cannot cope with variable manipulation [14] and abstract from implementation details [12], oWFNs cannot be used as an alternative rendering of BPEL processes.

After an IT expert modified the BPEL process towards execution, the business analyst may want to check whether the resulting BPEL process matches his business process model. This is either achieved by conformance checking techniques as presented in [15] or by mapping BPEL to BPMN [16]. Since BPELscript can be completely mapped to BPEL, the business analyst can use existing tools to check the conformance of the executable BPEL process with his process definition.

There is research whether visual programming or textual programming is better for program comprehension and for modifications of programs. In the case of structured flow-charts versus pseudocode, [17] shows that flow-charts are better for program comprehension. The experience report presented in [18] shows that "visual programming significantly reduces system development time". [19] states that "graphics may be better for technical, non-programmers than they are for programmers because of the great amount of experience that programmers have with textual notations in programming languages.". Finally, the studies presented in [20]–[22] show that "graphics [is] significantly slower than text" [23]. All in all, it is not finally proofed that visual programming is (in all cases) more suitable than textual programming. BPELscript and BPEL act on the same level, since both are textual descriptions. Currently, there is no research whether programmers are more effective using XML syntax or JavaScript-like syntax. For a comprehensive evaluation of a language, the development environment also has to be evaluated. The cognitive dimensions framework [24] provides a good basis for a comprehensive language evaluation. By presenting BPELscript this paper is a first step towards the comparison of BPEL and BPELscript.

IV. Enabling Graph-Oriented Programming

BPEL offers both, block-structured and graph-based programming [25]. Block-structured constructs are known from Java and other programming languages. In BPEL,

[2]http://www.eclipse.org/stp/b2j/

`sequence` is used to put activities in a sequence, `if` provides branching capabilities and loops can be expressed using `repeatUntil`, `while` and `forEach`. The `flow` activity is used for parallel processing. For graph-based programming, links can be explicitly defined within a `flow` activity to model synchronization. The link can be annotated with a transition condition. During the execution, the transition condition is evaluated and the status of the link set to the evaluation result. As soon as the status of all incoming links of an activity is defined, the join condition is evaluated. The join condition is a Boolean function over the status of the incoming links. Using join conditions, AND, OR and XOR joins can be modeled. If the join condition evaluates to true and the activity is executed. As soon as the activity is finished, the transition conditions of each outgoing link is evaluated and the result written as status to the link. The status of each outgoing link is set to false if the join condition evaluates to false. This execution semantics is called dead-path elimination (DPE) and is described for BPEL in [3], formalized in [26] and described in detail in [27].

Current structured programming languages do not have a first-class citizen to express control flow in terms of graphs. The structure and the current status of the graph is mainly stored in variables. Within the development of SimPEL, the Apache ODE team proposed two ideas to establish graph-oriented programming within a "structured program" approach [7], [28]. Main goals of these ideas are (i) to provide a scripting syntax inspired by JavaScript and Ruby and (ii) to combine the syntax of the fork/join parallelism concept with the semantic of BPEL. In summary, these approaches point to possible solutions to establish graph-oriented programming within a block-structured language:

(i) use an extended goto syntax (proposed by [28])
(ii) introduce new activities for forking and joining, with a new semantics (proposed by [7])

Option (i) is more close to BPEL. BPEL does not use explicit activities to fork and join, but uses properties of activities for that. In the XML serialization, these properties are realized as sub elements of the respective activity. For example, an AND join at an activity with two incoming links `l1` and `l2` is implemented by the attribute `joinCondition="$l1 and $l2"`. The attribute `joinCondition` can be put at any activity. There is no explicit AND join activity as it is the case at BPMN with the parallel gateway.

Option (ii) is more close to usual programming. In contrast to introduce new syntax elements, there are new statements with a certain semantics are introduced.

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. by using 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. Thus, we use a syntax being similar to activity statements, but translate these to sub elements of the belonging activity. A fork is expressed using `signal(<linkName>,<condition>);`. This is translated to a BPEL `link` annotated with the given link name and the given condition as transition condition as `source` child element of the preceding activity. Joining is expressed by `join(list of links, joinCondition);`. BPEL's default join condition is a logical or over all incoming links. This join condition can be overwritten using the `joinCondition` attribute. A `join` is translated to a `target` element of the subsequent activity. To illustrate the usage of `signal` and `join`, we show the representation the loan approval process in the next section.

It is important to note that representing graphs in a textual syntax is equivalent to programming using goto statements. Thus, the issues for goto programming [29] remain the same in the case of BPELscript. The aim of BPELscript is not to change the way business processes can be modeled with BPEL and to stay close to BPEL. Furthermore, `signal` and `join` enable parallelism. Thus, a transformation unstructured graphs to structured ones is not always possible [30].

V. Representing BPEL in BPELscript

BPELscript is designed to cover all features of BPEL. In this section, we outline the syntax of BPELscript and list the arguments which lead to the presented syntax.

Figure 5 presents the two parts of a BPELscript process: header and the process itself. The header contains namespace and import declarations. The `namespace` keyword is used to declare XML namespaces as the `xmlns` attribute does for XML [31]. `import` imports external types, such as WSDL message types or WSDL port types. WSDL stands for Web Services Description Language and is an interface definition language that defines the operations offered by a Web service [32]. BPEL distinguishes basic and structured activities. In general, structured activities are used to define the control flow structure and basic activities are the activities communicating with Web services and that manipulate data. In BPELscript basic activities are introduced with its activity keyword and are separated by a semicolon. For example `receive` is translated to `receive` and `invoke` to `invoke`. The only exception is the `assign` activity, which is directly translated to an assignment statement. The concrete syntax together with variable declarations is presented in Section V-B. Structured activities starts with its keyword too but are followed by an implicit sequence enclosed by single curly brackets. An explicit `sequence` construct is not introduced in BPELscript. Several statements in sequence are put in a BPEL `sequence` activity. For the `flow` activity SimPEL decided to provide `parallel` as keyword instead, since it sounds more familiar with concurrent execution [7]. We follow this argument and use `parallel` as translation for BPEL's `flow`.

A. Attributes

In BPEL, an activity can take mandatory and optional attributes. For example, the `operation` attribute is

```
// header
namespace* import*
// process
process tns::Name {
   activity;
}
```

// denotes a comment, * marks an element appearing zero or more times.

Figure 5. The structure of a BPELscript process

```
partnerLink plink =
  (ns::plType, roleName?, roleName?);
```

? marks an optional element.

Figure 6. Declaration of partner links

mandatory for the `invoke` activity, but the `name` attribute is optional. Mandatory attributes are put as parameters at the respective statement. Optional attributes are supported in BPELscript by using the annotation concept of Java. This means, that optional attributes are prefixed with an '@'-tag and placed in front of its corresponding statement.

B. Data handling

Partner links are used in BPEL to model a relationship between the BPEL process and a Web service. A partner link has a partner link type. The partner link type defines one or two roles and assigns one port type to each role. When instantiating a partner link type as partner link, one has to name the partner link type and state which role the process takes and which role the partner takes. To declare partner links, BPELscript uses a syntax similar to variable declarations in structured programming languages. Figure 6 presents the concrete syntax. `plink` is the name of the declared partner link. `ns::plType` is a reference to the partner link type to instantiate. The first `roleName` attribute denotes the role the business process takes. The second `roleName` attribute denotes the role of the partner. Role names can be omitted or set to `null` if the interaction is one-way only.

For the handling of process data, which are variables holding the state, BPELscript adopts the implicit variable declaration from SimPEL. In addition to implicit variable declaration, variables and partner links can be declared explicitly everywhere in a process.

Since BPELscript is block-structured, it provides assignments as usual with `lvalue=rvalue` [33]. As outlined in Figure 7, assignments appear in seven variants:

(i) literal assignment
(ii) reading from a message part
(iii) reading from a variable property
(iv) partner link assignments
(v) assignment from an expression
(vi) extended assignments using `copy`
(vii) extended assignments using or `extensionAssignOperation`

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"`.

In WSDL, a message has at least one part. If one such part has to be read, BPEL uses a dot as delimiter between

```
// literal assignment
my_var = "some value";

// reading from a message part
my_var = another_var.part;

// reading from a variable property
my_var = another_var#attr;

// partner link assignment
plink1 = plink2.myRole;
plink3 =
  "<sref:service-ref>...</sref:service-ref>"

// expressions
my_var = a + b;

// using XPath as expression language for rvalues
my_var = [bpel:doXslTransform
  ("urn:stylesheets:A2B.xsl", $A)];

// using an arbitrary language as
//  expression language for rvalues
my_var = [expression in language1]
  @queryLanguage="urn:language1";

// extended assign operation
{{{
  <extensionAssignOperation>
    <js:snippet>
      ... JavaScript/E4X code ...
    </js:snippet>
  </extensionAssignOperation>
}}}
```

Figure 7. Examples of assignments

the variable storing the message and the part. In BPELscript, we reuse this idea.

BPEL defines message properties to access a value in different types of variables. For example, the customer number may be located in the order message and in the order conformation message. To read from a property, the syntax `lvalue = variable#property` is used.

Assignments which affect partner links are similar to variables. In BPEL, the partner link variable stores the end point reference (EPR), which denotes where the partner can be reached. This EPR can be either read from the `myRole` or from `partnerRole`. A write to a partner link is always interpreted as write to the `partnerRole` element of a partner link. Therefore, `.partnerRole` is omitted in the lvalue of the partner link assignment. It is also possible to directly write EPRs to a partner link. In this case, the EPR has to be directly given as literal.

Expressions themselves follow the format defined by ECMAScript for XML (E4X, [34]). E4X is an official standard that adds direct support for XML to JavaScript. Thus, it is an ideal format to define expressions. The current implementation of the BPELscript translator only supports $+$, $-$, $*$ and $/$ as operators.

BPEL's default language for expressions in assignments is XPath 1.0 [35]. BPEL allows to change the expression language by using `queryLanguage`. This attribute is directly supported in BPELscript via the `@queryLanguage` annotation. In case BPEL's way of defining expressions is used, these expressions are enclosed by squared brackets as proposed by [7] and translated literally to BPEL.

In addition to changing the expression language for lvalues or rvalues, BPEL also allows an assign statement as a whole to be defined in a specified expression language. BPEL uses `extensionAssignOperation` as XML tag, which is reused in BPELscript. The XML element is nested in curly brackets, which denote that the content has to be literally translated. In case the extension assign operation is used, this feature has to be supported by the BPEL engine. In the case of E4X, the implementation is described in [36].

C. Fault handling

BPEL's construct to group activities together into a transactional unit is the `scope` activity. A `scope` activity provides fault, termination, compensation and event handling [3]. If an activity in a scope faults, the fault is delegated to a matching fault handler. A fault handler matches, if the name of the fault matches the name given in the fault handler. Otherwise, the fault is propagated one level up. If only fault handling should be specified for a group of activities, BPELscript offers the common `try {...} catch (faultName) |faultVariable|` syntax. The variable, where the fault information should be stored is specified via vertical bars. This simplified syntax is used in line 43 of Figure 4.

A scope is terminated if the enclosing scope encountered a fault or is terminated by itself. In this case, the termination handler is run. A scope may be compensated if it is completed. Compensation may only be started by a fault handler or compensation handler of the enclosing scope. Explicit compensation handling is defined in a compensation handler.

In business processes, it is important to wait at some point on an event [26]. An event may be a receipt of a message or the occurrence of a timeout. 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 event handlers. They are associated with its enclosing scope whose lifecycle determines the lifecycle of the handler. Event handlers are invoked when the corresponding event occurs.

Figure 8 presents the syntax of a scope in BPELscript. To offer a single way to specify fault handlers,there is no `catch` support at a `scope` statement: a `catch` has always to be specified together with a `try`.

VI. Translating BPEL to BPELscript and Vice Versa

As a proof of concept we implemented the "**B**PEL **to** BPEL**s**cript **T**ranslat**o**r" named *bosto*. The cornerstones of bosto are:

(i) to provide an extensible and fault-tolerant translation system which can be easily changed and
(ii) to use automated tools such as ANTLR [37] and ANTXR [38].

```
scope scopename {
  nop;
}
onCompensation { // compensation handler
  nop;
}
onTermination { // termination handler
  nop;
}
onEvents { // event handlers
  event(parterLink, operation) {
    nop;
  }
  timeout(p50d) {
    nop;
  }
}
```

Figure 8. Illustration of scopes in BPELscript

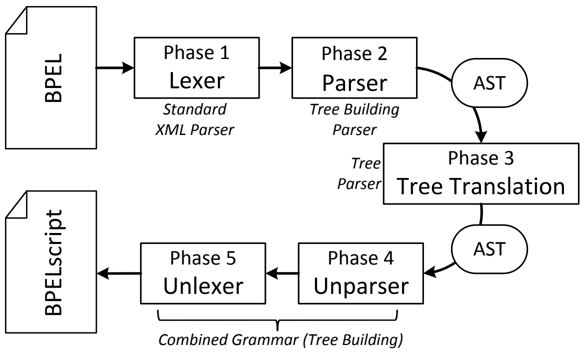

Figure 9. Translation process

ANTLR stands for"ANother Tool for Language Recognition" and is a tool that provides a framework for constructing language translators. ANTXR stands for **AN**other **T**ool for **X**ML **R**ecognition and extends ANTLR to support XML files.

A translation needs a broad understanding of the translation artifacts and has to cope with context. Therefore it is necessary to process a highly condensed version of the input which is called Abstract Syntax Tree (AST). The translation from BPEL to BPELscript is split into three parts:

(i) process an AST_a
(ii) translate AST_a into its counterpart AST_b
(iii) "unparse" the AST_b.

The whole process is shown in Figure 9. First, the BPEL process is parsed using a standard XML parser. ANTXR is used in phase 2 to build an AST of the BPEL process. A DOM tree cannot be used, since ANTLR does not support to specify tree translations for DOM trees, but for AST trees. In phase 3, the AST is translated to an AST representing the corresponding BPELscript AST. This AST is translated to a BPELscript file using a grammar combining the unparsing and unlexing of the BPELscript AST.

The translation from BPELscript to BPEL follows the same principle: first, BPELscript is lexed and parsed into an AST. This AST is then translated into a BPEL AST, which in turn is serialized in XML using a combined grammar.

Since the BPM lifecycle has the aim to continuously improve the process model, a process definition is never stable and is permanently adapted. Thus, the BPEL process has to be remodeled all the time and consequently the BPELscript representation will change accordingly. In order to keep the technical details that have been added during the executable completion within the configuration phase, the model cannot simply be regenerated but rather needs to be updated in a smart way. Therefore we take the original generated model $\mathcal{BS}_g$ and the model generated within the second lifecycle round $\mathcal{BS}'_g$ to compute the difference $\Delta(\mathcal{BS}_g, \mathcal{BS}'_g)$. The result is being translated into an editscript [39] Δ_E that describes a sequence of edit operations to transform $\mathcal{BS}_g$ to $\mathcal{BS}'_g$. Let $\mathcal{BS}_e$ be the executable completion of $\mathcal{BS}_g$. Now we apply Δ_E to $\mathcal{BS}'_g$ to get a starting point for $\mathcal{BS}'_e$ that contains both, the new semantics of the process model and the refinements made in the previous lifecycle round. Note that it may be possible that not all editscript operations can be applied to the new model in case the model has significantly changed. This approach, however, makes it easier to cope with model changes during the configuration phase. As a side effect, the formatting and program organization made by the developer in the first round are kept. A detailed discussion of applying differences to models is discussed in [40].

VII. Conclusion and Future Work

We presented a new syntax of BPEL called BPELscript and sketched the integration in the BPM lifecycle. The syntax of BPELscript is not derived by simply replacing the opening and closing XML tags by opening and closing curly brackets: In the case of variables, BPELscript allows implicit variable declarations. The possibilities for data manipulation of BPEL are replaced by an E4X syntax. BPEL supports optional and mandatory attributes, which are reflected as annotations (@) and parameters for functions respectively. Finally, we discussed the reflection of control links offered by the `flow` activity as separate statements instead of sub-elements of a statement.

The full syntax of BPEL script and the translator is described in [41]. An online version of the presented translator is available at http://www.bpelscript.org, the full source code is available at http://code.google.com/p/bpelscript/. The current translation covers all important cases and shows that the idea of a two-way translation works. Ongoing work is to provide an IDE for BPELscript editing including syntax highlighting and online syntax check using the Eclipse Dynamic Languages Toolkit (DLTK[3]). This enables BPELscript to be evaluated using the cognitive dimensions framework [24].

The next step in BPELscript development is to develop a type system to ensure proper explicit variable declaration in BPEL of implicitly declared variables in BPEL script. In parallel, we are going to assess the acceptance of BPELscript in the industry and prove the effectiveness of BPELscript in comparison to directly edit XML code.

[3]http://www.eclipse.org/dltk/

Acknowledgments

This work is supported by the BMBF funded project Tools4BPEL (01ISE08B) and by the SUPER project[4] under the EU 6th Framework Programme Information Society Technologies Objective (contract no. FP6-026850).

References

[1] M. Weske, *Business Process Management: Concepts, Languages, Architectures*. Springer-Verlag, 2007.

[2] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. F. Ferguson, *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More*. Prentice Hall PTR, 2005.

[3] OASIS, "Web Services Business Process Execution Language (WS-BPEL) – Version 2.0," 2007, http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html.

[4] *Business Process Modeling Notation, V1.2 – OMG Available Specification*, Object Management Group, Jan. 2008. [Online]. Available: http://www.omg.org/spec/BPMN/1.2/PDF

[5] C. Ouyang, M. Dumas, S. Breutel, and A. ter Hofstede, "Translating Standard Process Models to BPEL," in *Proceedings 18th International Conference on Advanced Information Systems Engineering (CAiSE)*, ser. Lecture Notes in Computer Science, vol. 4001. Springer-Verlag, June 2006.

[6] G. Bischoff, R. Kersten, and T. Vetter, "Vergleich von BPEL-Workflow Modellierungstools," Student Report Software Engineering, IAAS, Universität Stuttgart, May 2005, (in German). [Online]. Available: http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=FACH-0038&engl=1

[7] A. Boisvert, A. Arkin, and M. Riou, "BPEL SimplifiedSyntax (simPEL)," 2008. [Online]. Available: http://ode.apache.org/bpel-simplified-syntax-simbpel.html

[8] I. Gavran, A. Milanovic, and S. Srbljic, "End-User Programming Language for Service-Oriented Integration," in *7th Workshop on Distributed Data and Structures*, 2006.

[9] D. Skrobo, A. Milanovic, and S. Srbljic, "Distributed Program Interpretation in Service-Oriented Architectures," in *9th World Multi-Conference on Systemics, Cybernetics and Informatics (WMSCI 2005)*, 2005.

[10] F. v. Breugel and M. Koshkina, "Models and Verification of BPEL," 2006. [Online]. Available: http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf

[11] P. Massuthe, W. Reisig, and K. Schmidt, "An Operating Guideline Approach to the SOA," *Annals of Mathematics, Computing & Teleinformatics*, vol. 1, no. 3, pp. 33–43, 2005.

[12] N. Lohmann and J. Kleine, "Fully-automatic Translation of Open Workflow Net Models into Simple Abstract BPEL Processes," in *Modellierung*, ser. Lecture Notes in Informatics, vol. P-127. Gesellschaft für Informatik e. V., 2008.

[13] W. M. P. van der Aalst, "The application of Petri nets to workflow management," *Journal of Circuits, Systems and Computers*, vol. 8, no. 1, pp. 21–66, 1998.

[14] N. Lohmann, "A Feature-Complete Petri Net Semantics for WS-BPEL 2.0," in *4th International Workshop on Web Services and Formal Methods*, ser. Lecture Notes in Computer Science, vol. 4937. Springer-Verlag, 2007.

[15] A. Martens, "Consistency between Executable and Abstract Processes," in *2005 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'05)*. IEEE Computer Society, 2005.

[4]http://www.ip-super.org

[16] M. Weidlich, G. Decker, A. Großkopf, and M. Weske, “BPEL to BPMN: The Myth of a Straight-Forward Mapping,” in *16th International Conference on Cooperative Information Systems (CoopIS)*, ser. Lecture Notes in Computer Science, vol. 5331. Springer-Verlag, 2008.

[17] D. A. Scanlan, “Structured Flowcharts Outperform Pseudocode: An Experimental Comparison,” *IEEE Software*, vol. 6, no. 5, pp. 28–36, Sep 1989.

[18] E. Baroth and C. Hartsough, “Visual programming in the real world,” in *Visual object-oriented programming: concepts and environments*. Manning Publications Co., 1995.

[19] J. D. Kiper, B. Auernheimer, and C. K. Ames, “Visual Depiction of Decision Statements: What is Best forProgrammers and Non-Programmers?” *Empirical Software Engineering*, vol. 2, no. 4, pp. 361–379, 1997.

[20] T. R. G. Green, M. Petre, and R. K. E. Bellamy, “Comprehensibility of visual and textual programs: a test of superlativism against the ’match-mismatch’ conjecture,” in *Empirical Studies of Programmers, Fourth Workshop*. Open University, Computer Assisted Learning Research Group, 1991.

[21] T. R. G. Green and M. Petre, “When visual programs are harder to read than textual programs,” in *Sixth European Conference on Cognitive Ergonomics (ECCE-6)*, 1992.

[22] T. Moher, D. Mak, B. Blumenthal, and L. Leventhal, “Comparing the comprehensibility of textual and graphical programs: The case of Petri nets,” in *Empirical Studies of Programmers: Fifth Workshop*. Ablex, 1993.

[23] M. Petre, “Why looking isn’t always seeing: readership skills and graphical programming,” *Commun. ACM*, vol. 38, no. 6, pp. 33–44, 1995.

[24] T. R. G. Green and M. Petre, “Usability Analysis of Visual Programming Environments: A ‘Cognitive Dimensions’ Framework,” *Journal of Visual Languages & Computing*, vol. 7, no. 2, pp. 131 – 174, 1996.

[25] O. Kopp, D. Martin, D. Wutke, and F. Leymann, “The Difference Between Graph-Based and Block-Structured Business Process Modelling Languages,” *Enterprise Modelling and Information Systems*, vol. 4, no. 1, pp. 3–13, June 2009.

[26] F. Leymann and D. Roller, *Production workflow: concepts and techniques*. Prentice Hall PTR, 2000.

[27] F. Curbera, R. Khalaf, F. Leymann, and S. Weerawarana, “Exception Handling in the BPEL4WS Language,” in *International Conference on Business Process Management (BPM)*, ser. Lecture Notes in Computer Science, vol. 2678. Springer-Verlag, 2003.

[28] T. van Lessen and O. Kopp, “BPEL Simplified Syntax (simBPEL),” 2008. [Online]. Available: http://ode.apache.org/bpel-simplified-syntax-simbpel.html

[29] E. W. Dijkstra, “Letters to the editor: go to statement considered harmful,” *Commun. ACM*, vol. 11, no. 3, pp. 147–148, 1968.

[30] B. Kiepuszewski, A. H. M. ter Hofstede, and C. J. Bussler, “On Structured Workflow Modelling,” in *Advanced Information Systems Engineering*, ser. Lecture Notes in Computer Science, vol. 1789. Springer Berlin / Heidelberg, 2000, pp. 431–445. [Online]. Available: http://www.springerlink.com/content/r3b8597wgqgjpy6r/

[31] *Namespaces in XML 1.0 (Second Edition)*, W3C, Aug. 2006. [Online]. Available: http://www.w3.org/TR/xml-names/

[32] *Web Services Description Language (WSDL) 1.1*, W3C, Mar. 2001. [Online]. Available: http://www.w3.org/TR/2001/NOTE-wsdl-20010315

[33] S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[34] *Standard ECMA-357—ECMAScript for XML (E4X) Specification*, Ecma International, 2005. [Online]. Available: http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-357.pdf

[35] *XML Path Language (XPath) Version 1.0*, W3C, Nov. 1999, http://www.w3.org/TR/xpath.

[36] T. van Lessen, J. Nitzsche, and D. Karastoyanova, “Facilitating Rich Data Manipulation in BPEL using E4X,” in *ZEUS: Zentraleuropäischer Workshop über Services und ihre Komposition*, vol. 438. CEUR Workshop Proceedings, 2009.

[37] T. Parr, *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Raleigh: The Pragmatic Bookshelf, 2007.

[38] S. Stanchfield *et al.*, “ANTXR: Easy XML Parsing, based on the ANTLR parser generator, Website.” [Online]. Available: http://www.antlr.org/

[39] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, “Change detection in hierarchically structured information,” in *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Assn for Computing Machinery, Jun. 1996, pp. 493–504.

[40] E. Kindler, P. Könemann, and L. Unland, “Difference-based model synchronization in an industrial MDD process,” in *2nd ECMDA Workshop on Model-Driven Tool & Process Integration (MDTPI 2009)*, 2009.

[41] M. Bischof, “Translating WS-BPEL 2.0 to BPELscript and Vice Versa,” Studienarbeit, IAAS, Universität Stuttgart, 2008. [Online]. Available: http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=STUD-2175&engl=1

All links were followed on 2009-05-27.