Verifying and Testing BPEL Processes

Frédéric Servais

Directeur: Prof. Esteban Zimányi
Codirecteur: Prof. Jean-François Raskin

Mémoire présenté en vue de l’obtention
du Diplôme d’Étude Approfondie
en Sciences Appliquées
Année académique 2006–2007
Contents

1 Introduction .................................................. 4
  1.1 Web Services Stack ...................................... 5
  1.2 BPEL ....................................................... 6
    1.2.1 Coordination ........................................ 6
    1.2.2 Composition and Workflow ............................ 7
  1.3 Verification and Testing of BPEL Processes .............. 9

2 Web Services Basics ...................................... 11
  2.1 XML - eXtensible Markup Language ...................... 11
  2.2 XML Schema .............................................. 12
  2.3 WSDL - Web Services Description Language ............. 16
    2.3.1 Types ............................................... 17
    2.3.2 Messages ............................................ 17
    2.3.3 Port Types and Operations .......................... 18
    2.3.4 Example: The Loan Assessor ......................... 19
  2.4 XPath ...................................................... 20
  2.5 Conclusion ............................................... 21

3 BPEL ......................................................... 22
  3.1 Example - Loan Approval Service ....................... 22
  3.2 Overview of BPEL Language ................................ 28
    3.2.1 Process .............................................. 28
    3.2.2 Variables ........................................... 29
    3.2.3 Expressions ......................................... 29
    3.2.4 Basic Activities .................................... 30
    3.2.5 Structured Activities ............................... 32
    3.2.6 Links ............................................... 35
  3.3 Conclusion ............................................... 36
4 Model Checking BPEL Services 37
   4.1 Petri Nets model .............................................. 38
      4.1.1 Mapping BPEL into Petri Nets .......................... 38
      4.1.2 Static Analysis of BPEL Processes ...................... 43
   4.2 Guarded Automata Model .................................... 45
      4.2.1 Guarded Automata Peers Model .......................... 46
      4.2.2 LTL Properties ........................................... 47
      4.2.3 From BPEL to Guarded Automata to Promela .......... 48
      4.2.4 Results .................................................. 52
   4.3 Conclusions .................................................. 53

5 Testing BPEL Web Services 54
   5.1 Test Suites .................................................. 54
   5.2 Generating Test Cases using SPIN for BPEL Services ...... 55
      5.2.1 BPEL to Promela ......................................... 56
      5.2.2 SPIN to Test Cases ...................................... 58
      5.2.3 Results .................................................. 58

6 Conclusions and Future Works 60
Acknowledgment

I hereby express my deepest gratitude to my supervisor Professor Esteban Zimanyi and my co-supervisor Professor Jean-Francois Raskin for supporting this work all over the year. I also warmly thank Koen, Ivan, Candide and last but certainly not least Jean-Michel for their invaluable help.
Chapter 1

Introduction

Contents

1.1 Web Services Stack ............................... 5
1.2 BPEL ........................................... 6
   1.2.1 Coordination ................................. 6
   1.2.2 Composition and Workflow ................. 7
1.3 Verification and Testing of BPEL Processes ........ 9

When Internet first developed, it was meant to be used for the sharing of information. Soon, client and server applications communicating through the web appeared. While html-based web applications and applets were the visible part of that trend, the business world invested heavily in getting their inter-enterprise application integrated in order to streamline their operations. The need to unite around universally accepted standards for system interaction through the internet quickly materialized.

Supervised by the World Wide Web Consortium and other associations like OASIS, the standardisation effort involves the whole industry as well as numerous academics. After more than 10 years many standards have been proposed: XML, HTML, HTTP, FTP, DTD, XML Schema, XPath, SOAP, WSDL, WS-BPEL, WSTransaction, ... The emergence of this wide range of broadly accepted standards enables the reengineering of the whole intra and inter-enterprise application integration. The Web Services technologies play a central role in this trend to global businesses integration.
1.1 Web Services Stack

According to the W3C’s Web Services Architecture Working Group, a Web Service is “a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards” [13]. Web services can thus be viewed as a stack of standards (see Figure 1.1).

![Web Services Architecture Stack](image)

Figure 1.1: Web Services Architecture Stack [13].

At the very bottom of this stack we find the communication standards that are already used for every single interaction through the Web: http, smtp, ftp,... They are responsible for transporting messages between applications.

The base technologies constituted the next layer of the stack: XML, DTD, XML Schema, XPath, XQuery,... They are used for the description and manipulation of the information appearing throughout the Web Services
Applications will communicate through messages formatted in XML. They will use a message exchange protocol like OASIS’s SOAP standard or XML-RPC or other alternatives like REST.

These messaging protocols will be used and associated with Interface Definition Languages, such as the Web Service Definition Language (WSDL), to provide a machine readable definition of the offered service. A WSDL document is essentially a bare list of the operations supported by the Web Service.

These layers conclude the basic Web Services infrastructure upon which the other, higher level standards will build. Higher level standards include: WS-Transaction for interactions requiring long running transaction management, WS-Federation for grouping disparate user and machine identities across heterogeneous systems, WS-Provisioning to facilitate interoperability between different provisioning systems, . . .

Business Process Execution Language for Web Services (BPEL) is another of these higher level specifications.

1.2 BPEL

BPEL is a coordination and composition language for Web Services. It can also be viewed as a workflow language for Web Services. We present in more details these notions now.

1.2.1 Coordination

The standards defined above (WSDL, SOAP, HTTP . . . ) enable the implementation of basic Web Services. They permit in particular interactions where the client invokes a single operation.

New standards are needed if we want to support interactions involving several invocations. Indeed the sequences of invocations necessary to use a service must usually follow a given order and comply with business rules. The client and the server will have to obey these constraints.

Examples A travel service will require the customer firstly to reserve his flight before actually paying or canceling it. The payment nor the cancellation can be done before the reservation. An online shop will require the customer to add at least one item to his shopping cart before checking out.

These rules can be documented informally and then manually translated into the programming language used on the client and on the server. A
better approach is to use a declarative language to define the interaction rules, just as WSDL defines the interface of a web service. Having a standard, declarative language to define these interactions makes it possible to attach these service constraints to the definition of the service. Programmers can make use of these definitions, easing the development of compliant clients.

The term conversation denotes a sequence of invocations that occur between the client and the server during a web service interaction. The term coordination protocol, or business protocol is used to denote the specification of the set of valid conversations. The coordination is focused on the external perspective of the interaction involving several invocations, nothing is being said about how to implement the mechanism to conduct these conversations (client and server side).

A BPEL process can define a coordination protocol, using the notion of abstract process.

### 1.2.2 Composition and Workflow

The relevance of coordination pertains to the external view of the interactions. Composition concerns the internal part of the mechanisms implementing these compliant conversations.

A composite service is a Web Service implemented by combining several other Web Services\(^\text{1}\). The process of developing a composite service is called Web Service composition.

**Example: The Loan Approval Service.** This service provides a Web Service port to which customers can send their loan requests. A loan request includes personal information of the customer and the desired amount. The automated service must send a reply to the customer with an ‘approved’ or ‘rejected’ message, it proceeds as follows.

First, It checks if the amount is greater than $10,000. If it is, then the Loan Approval Web Service will be invoked by forwarding the request, passing the personal information and the amount requested. When called, the Loan Approval service takes a final decision that will be transmitted to the customer. If the amount requested by the customer below $10,000, a procedure is invoked on the Risk Assessment Web Service. If the Risk Assessment service tags the request as ‘low’ risk, then the request is automatically approved. If the Risk Assessment service tags the customer request as ‘high’

\(^{1}\)Actually it is a combination of some of the operations offered by the involved Web Services.
Figure 1.2: The Loan Approval Service.
risk then the request is forwarded to the Loan Approval Web Service that takes the final decision.

The Loan Approval Service is a Web Service obtained by composition of two Web Services: the Risk Assessment Web Service and the Loan Assessment Web Service. A more complex Web Service is here built from simpler ones.

The coordination protocol imposes constraints on the composition, since the order of composition must be compliant with the rules defined by the coordination protocol. On the other hand, the internal composition defines the conversations the service (client or server) is able to execute. There is, thus, a tight relationship between the internal composition and the external coordination of a Web service.

A BPEL process can define a composition of Web Services using the notion of executable process.

**Workflow**

When a customer makes a loan request to the Loan Approval Service, we say that a new case has started. This case will flow through the business process, i.e. the decision making procedure, presented above. Evolution of a case is discrete in nature, it has a beginning and an end, and can be uniquely identified.

A workflow, or business process, consists of a number of tasks that must be performed and a set of conditions that determine the order and nature of these tasks. A task is a logical unit of work. In the Loan Approval Process the tasks are the reception of a request, the risk assessment, the loan assessment and the reply to the customer. There are two conditions. The first is based on the comparison of the requested amount and '$10,000', the second is based on the risk level.

In [23], the authors identify four basic constructs of workflow structure: sequence, selection, iteration and parallelization. All processes can be, in principle, modeled with these basic elements.

BPEL can be viewed as a language to define workflows, where the tasks are basic Web Service operations, like receive, invoke or reply.

### 1.3 Verification and Testing of BPEL Processes

The need to verify and validate BPEL processes is obvious. As with any other software system, validating the implementation against the expected behav-
ior is essential. Two popular methods are available for software verification and validation: model checking and testing.

Model Checking [14] is a set of techniques for verifying properties of formal systems through modeling. Properties are often specified in Linear temporal Logic (LTL) or Computation Tree Logic (CTL).

The system to be verified is first transformed into a model, e.g. a finite state machine. The desired properties of the system are expressed as properties of the model in such a way that if the properties hold in the model then it will hold in the system. Or alternatively, if the property does not hold in the model then it should be possible to deduce an execution of the system leading to the violation of the original property.

The use of model checking and formal methods helps reduce the number of errors in the early stages of development of a software. In the later stages, testing is a popular method for verifying the correctness of the implementation against its expected behavior.

Testers usually write test cases. A test case or unit test is the description of an input on which the software is to be executed. A test case can also contain the expected result or response from the program. The program passes the test if the actual response is the response that was expected. A set of test cases is called test set or test suite.

In this paper we explore the results of the application of model checking techniques to the verification and testing of BPEL processes. In Chapter 2, we present the standards BPEL is relying on: XML, XML Schema, WSDL and XPath. In Chapter 3, we introduce the main concepts of BPEL. In Chapter 4, some important results on the verification of BPEL processes are presented. Finally in Chapter 5, we outline what has been done in the field of BPEL process testing based on model checking techniques.
Chapter 2

Web Services Basics

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>XML - eXtensible Markup Language</td>
<td>11</td>
</tr>
<tr>
<td>2.2</td>
<td>XML Schema</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>WSDL - Web Services Description Language</td>
<td>16</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Types</td>
<td>17</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Messages</td>
<td>17</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Port Types and Operations</td>
<td>18</td>
</tr>
<tr>
<td>2.3.4</td>
<td>Example: The Loan Assessor</td>
<td>19</td>
</tr>
<tr>
<td>2.4</td>
<td>XPath</td>
<td>20</td>
</tr>
<tr>
<td>2.5</td>
<td>Conclusion</td>
<td>21</td>
</tr>
</tbody>
</table>

In this chapter we present the main technologies BPEL is based on: XML, XML Schema, WSDL and XPath.

2.1 XML - eXtensible Markup Language

The Extensible Markup Language [26], known as XML, version 1.1 is a W3C Recommendation since February 2004. XML is a worldwide standard defining a way to encode structured data. A document following the XML specification is called an XML document. We very briefly, with an example, expose the basis of an XML document.

An element is the basic construct of an XML document. It is composed of an opening (<elementName>) and a closing tag (<elementName>). An XML document is a set of elements grouped in a tree structure, it has thus
exactly one root element. Elements can have child elements and attributes. The data is located within the element itself or one of its attribute.

Example

```xml
<recipe name="bread" preptime="5 mins" cooktime="3 hours">
  <title>Basic bread</title>
  <ingredient amount="3" unit="cups">Flour</ingredient>
  <ingredient amount="0.25" unit="ounce">Yeast</ingredient>
  <instructions>
    <step>Mix all ingredients together.</step>
    <step>Knead, place in a tin then bake in the oven.</step>
  </instructions>
</recipe>
```

The `recipe` element is the root element, it has 4 child elements: `title`, `ingredient`, `ingredient` and `instructions`. It has 3 attributes: `name`, `preptime` and `cooktime`.

An XML document is said to be well-formed if it complies with all XML’s syntax rules.

### 2.2 XML Schema

XML Schema [30], a W3C Recommendation, is a language for the description of a family, or type, of XML documents. An XML Schema, schema for short, is an XML document composed of a set of rules, that apply restrictions to elements, attributes and their content. An XML document is valid against a given schema if it conforms to its rules. While the well-formedness was a syntactic property, the validity of an XML document against a schema is a property about its structure and its content.

The schema first defines a set of types, simple and complex, which are then used to define the valid XML structures by giving each possible element a type. We very briefly review, with a simple example, the main concepts of XML Schema.

XML Schema has a powerful data type definition mechanism. It differentiates between simple types and complex types. Elements that have children and/or that have attributes are said to be of complex type. While attributes and elements without children and attributes are said to be of simple type.

#### Simple types

XML Schema has a collection of built-in simple types. The types are represented in the Figure 2.1 below, it includes: `date`, `int`, `boolean`, `string`, `float`,...
Figure 2.1: XML Schema data types [30].
Simple types can be derived from these built-in ones. The element `simpleType` declares a new simple type, it will be referred to with its name from the `name` attribute. In the example below, a new type `nameType` is declared and is of type `string` with a maximum of 32 characters. The type `isbnType` declares a `string` type that must be composed of 10 digits.

```xml
<xs:simpleType name="nameType">
  <xs:restriction base="xs:string">
    <xs:maxLength value="32"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="isbnType">
  <xs:restriction base="xs:string">
    <xs:pattern value="[0-9]{10}"/>
  </xs:restriction>
</xs:simpleType>
```

Complex type definition There is no built-in complex type. Complex types are defined with the `complexType` element. There are many available constructs to define complex types: the `simpleContent` Schema element is used to declare types of elements containing data within themselves, `element` declares a child, `attribute` Schema element declares an attribute and its type (limited to simple type), `sequence` Schema element declares an order list of subelements, `minOccurs` and `maxOccurs` Schema attributes are used to express the number of occurrences the concerned element can have, and so on.

In the example below, a purchase order is declared. A valid element against that Schema type must have an attribute `orderDate` conforming to the `date` Schema type, and has an ordered list of children: a `shipTo` element, a `billTo` element, an optional `comment` element and one `items` element, all of their respective types.

```xml
<xsd:complexType name="PurchaseOrderType">
  <xsd:sequence>
    <xsd:element name="shipTo" type="USAddress"/>
    <xsd:element name="billTo" type="USAddress"/>
    <xsd:element ref="comment" minOccurs="0"/>
    <xsd:element name="items" type="Items"/>
  </xsd:sequence>
  <xsd:attribute name="orderDate" type="xsd:date"/>
</xsd:complexType>
```
Element definitions  Element definition enforces structure by putting elements in hierarchical relation and giving them a type. The Schema element used for this is `element`.

Example

```xml
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xs:simpleType name="nameType">
        <xs:restriction base="xs:string">
            <xs:maxLength value="32"/>
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="isbnType">
        <xs:restriction base="xs:string">
            <xs:pattern value="[0-9]{10}"/>
        </xs:restriction>
    </xs:simpleType>

    <xs:element name="book">
        <xs:complexType mixed="true">
            <xs:sequence>
                <xs:element name="title" type="xs:string"/>
                <xs:element name="author" type="nameType"/>
            </xs:sequence>
            <xs:attribute name="isbn" type="isbnType"/>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

A valid document against that schema is the following:

```xml
<book isbn="0836217462">
    <title>
        Being a Dog Is a Full-Time Job
    </title>
    <author>Charles M. Schulz</author>
</book>
```
2.3 WSDL - Web Services Description Language

The Web Services Description Language [28], known as WSDL, is an XML document format for describing Web Services. The current version, WSDL 1.1, is a W3C note but was not endorsed as a W3C recommendation. Nevertheless, version 2.0 is expected to become a W3C recommendation in the near future.

A WSDL document is composed of seven elements:

Types definition of data types used in this service.
Message definition of messages involved in the interactions with this service.
Operation abstract operations.
Port Type abstract group of operations supported by the service.
Binding concrete protocol and data format for a port type of this service.
Port binding and network address.
Service a collection of ports.

Types, Message, Operation and Port Type belong to the abstract part of the service description. The abstract part is clearly separated from the concrete part, i.e. Binding, Port and Service, to allow reuse of the abstract description.

A WSDL document has the following structure:

```xml
<definitions name="...">
  <types>
    [...]
  </types>
  <message name="...">
    [...]
  </message>
  <portType name="...">
    [...]
  </portType>
  <binding>
    [...]
  </binding>
  <service>
    [...]
  </service>
</definitions>
```
In this document we are exclusively concerned with the abstract part of
the description. We now describe the types, messages, operations and port
type declarations.

2.3.1 Types
WSDL does not have a standard for data type definition, instead it is designed
to be very flexible and has a mechanism for plugging in any data type system.
WSDL does however use the XML Schema data type system by default (see
Section 2.2). An WSDL engine implementation must support this data type
system in order to be WSDL compliant.

The types element enables the declaration of types into a WSDL doc-
ument. In the sample below an array of string is defined in the standard
XML Schema way.

```xml
<types name="StringArray">
  <xsd:schema targetNamespace="http://typeSample.org">
    <xsd:complexType name="string_array">
      <xsd:choice maxOccurs="20">
        <xsd:element name="item" type="xsd:string"/>
      </xsd:choice>
    </xsd:complexType>
  </xsd:schema>
</types>
```

2.3.2 Messages
Messages are the units of communication for Web services, they are used as
input, output and reporting of faults of the operations. A message has a
name and is a flat structure composed of ‘parts’. Each part has a name and
a type declared in the type element of the document.

In the example below, messages creditInformationMessage, loanRequestErrorMessage, riskAssessmentMessage and approvalMessage
are defined. The involved types must be defined in the types element of the
document.

```xml
<message name="creditInformationMessage">
  <part name="firstName" type="nameType"/>
  <part name="name" type="nameType"/>
  <part name="amount" type="amountType"/>
</message>
<message name="loanRequestErrorMessage">
```
2.3.3 Port Types and Operations

Port types regroup operations under a port type name. An operation can be seen as a function in the C programming language. There are four types of operation:

**One-way.** The service receives a message.

**Request-response.** The service receives a message, and sends a correlated message.

**Solicit-response.** The service sends a message, and receives a correlated message.

**Notification.** The service sends a message.

Operation definitions require a name, input parameters, an output type and the errors that can be raised during processing of the request. The input and output types are messages defined in the message element of the document. Depending on the type of operation the input, output and fault messages can or not be present.

The example below illustrates the declaration of two port types, the loanApprovalPT and the riskAssessmentPT. Each contains a single operation using messages (for input, output and fault) defined in the message section 2.3.2. The riskAssessmentPT port type has one check operation that takes a creditInformationMessage as input and sends a riskAssessmentMessage as response.
2.3.4 Example: The Loan Assessor

A WSDL document starts with the definitions root element in which the namespace aliases that will be used in the document are usually declared. Subsequently, the types, messages, port types with operations, bindings and services section are declared.

Putting all the above samples together, and adding some type definitions, we obtain the WSDL definition of the loan approval service. It defines a service with one port type, the riskAssessmentPT defined in Section 2.3.3.

```xml
<definitions
    targetNamespace="http://tempuri.org/services/loanassessor"
    xmlns:tns="http://tempuri.org/services/loanassessor"
    xmlns:loandef="http://tempuri.org/services/loanassessor"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wSDL/soap/"
    xmlns="http://schemas.xmlsoap.org/wSDL/">

    <!-- types declaration-->

    <!-- messages declaration-->
    <message name="riskAssessmentMessage">
        [...]
    </message>

    <!-- port type declaration-->
    <portType name="riskAssessmentPT">
        [...]
    </portType>

    <!-- concrete section-->
    <service name="LoanAssessor">
        ...
    </service>
    <binding name="riskAssessmentSoap" type="riskAssessmentPT">
        <soap:binding style="document"
            transport="http://example.com/http"/>
        <operation name="assess">
```
The bindings define the concrete message formats, protocol details for operations and messages defined by a particular port type. A port defines an individual endpoint by linking a binding to an address. Finally, a service groups a set of related ports together. The most popular binding practice is with the SOAP protocol [27]. WSDL includes a binding for SOAP 1.1 endpoints.

2.4 XPath

XPath [29], a W3C Recommendation since the 16th November 1999, is a query and expression language for XML documents. Originally, XPath aimed at providing a common syntax for functionalities shared between XSL Transformations (XSLT) and XPointer (XPointer), however it quickly evolved into an independent language for querying XML documents.

The central construct of XPath is the expression. The evaluation of an expression on an XML document yields to an XML node, a set of XML nodes, a number, a string or a boolean value. The evaluation of the expression depends on a context. The context contains the XML document, a node, called the context node or current node, and other context informations. We illustrate XPath with a few examples.

A path expression returns a node or a set of nodes. Examples:

- **author** - returns the set of child nodes of the context node named author.
- ***/ - returns the set of all children of the current node.
• @name - returns the attribute named name of the current node.


• //recipe/ingredients - returns the all ingredients direct children of the root node recipe.

• chapter[@number="3"] - returns the chapters whose attribute number is 3.

• employee[@secretary and @assistant] - returns all the employee children of the context node that have both a secretary attribute and an assistant attribute.

General expressions return number, string or boolean values:

• 3.141529 - numeric constant.

• 2+2 - additive expression.

• 'All I Have Is Now' - string constant.

• true - boolean constant.

• //recipe/ingredient[1][@quantity] <5 - the first ingredient’s quantity is less than 5.


2.5 Conclusion

We presented in this chapter the four technologies BPEL depends on. XML defines the format of any information. XML Schema provides a versatile data type system. WSDL is the standard for defining Web Service interfaces. XPath is a simple expression language for XML documents. BPEL really builds upon these four standards.
Chapter 3

BPEL

Contents

3.1 Example - Loan Approval Service .................................. 22
3.2 Overview of BPEL Language ........................................ 28
  3.2.1 Process ....................................................... 28
  3.2.2 Variables .................................................. 29
  3.2.3 Expressions ............................................... 29
  3.2.4 Basic Activities ........................................... 30
  3.2.5 Structured Activities ..................................... 32
  3.2.6 Links ....................................................... 35
3.3 Conclusion .......................................................... 36

3.1 Example - Loan Approval Service

We illustrate BPEL with the Loan Approval Web Service (see ).

There are three different services involved in an interaction for the loan
approval service. The loan approval service implements the loanServicePT
port type defined below. The risk assessor implements riskAssessmentPT
port type and the loan approver the loanApprovalPT port type.

```xml
<portType name="loanServicePT">
  <operation name="request">
    <input message="lns:creditInformationMessage"/>
    <output message="lns:approvalMessage"/>
    <fault name="unableToHandleRequest"
          message="lns:errorMessage"/>
  </operation>
</portType>
```

22
To each of these port types corresponds a partner link type. A partner link type is the association of a port type with roles. The roles are played by partners. However in this case the interaction are one way. Indeed, the loan service never invoke an operation of the customer and the loan assessor nor the risk assessor invoke an operation of the loan service. Thus each partner link just involved one partner. Partner link are needed to define a BPEL process.

There are four different types of message involved in the interaction with the Loan Approval Service. The creditInformationMessage sent by the customer and forwarded to the loan assessor, approvalMessage reply of the loan approval service (to the customer), riskAssessmentMessage in reply from the risk assessment service and errorMessage for all the interaction in case of failure.
These message types, port types and partner links must be defined in the appropriate WSDL documents. We can now define the BPEL process for the Loan Approval Service.

**Process** Each BPEL document starts with the `process` element in which one usually declares the namespace that will be used in the document. In this case the default namespace refers to the BPEL namespace, `lns` refers to the appropriate WSDL definitions.

The first step is to declare the parties involved in the interaction. In this case there are four parties: the loan service, the customer, the approver and the assessor. We refer to the loan service as ‘the service’, while the other parties are the partners. Each partner is declared in the `partner` element, child of the `process` element. A partner is given a name by which it will be referred in the rest of the BPEL document and is associated to the WSDL interface it is assumed it complies to.

In the example below, the customer is involved in the loanApprovalLink as a client of this process which plays the approver, i.e. this process must comply to the loanApprovalPT which has only one operation: approve. The approver partner is involved in the same link type but it plays the role of the approver and this process plays the client (will invoke the approve operation of the approver). Finally the assessor plays the assessor in an riskAssessmentLink in which this process plays the client.
The second steps is to declare, in the variables element, the variables that are local to the process. These variables persist the state of the interaction. Each of the variable declared in a variable element is accessible during the whole life cycle of the process. These variables are identifiable with their names. Their types must be one declared in a WSDL message definition.

Now everything is in place to actually define the business process in itself. The process is defined as a flow. The flow construct permits launching several activities concurrently. The links defined in the flow scope permits to enforce precedence between these activities, i.e. it permits synchronization. In other words the flow construct associated with links permits defining an activity diagram.

The process, as illustrated by the diagram in Figure 3.1, starts when the approve operation of the service is invoked, this is declared in the first receive element named receive1. Once this call is received, the link receive-to-assess is activated if the corresponding condition is true, else the receive-to-approval link is activated. Thus accordingly the assessor or the approver will be invoked. If invoked, the approver will trigger the reply. The assessor will invoke the approver that will trigger the reply or will activate the assign activity that will trigger the reply.

The actual BPEL definition is little more than the translation of the diagram above with BPEL construct.

```xml
<partner name="customer"
   serviceLinkType="lns:loanApprovalLinkType"
   myRole="approver"/>
<partner name="approver"
   serviceLinkType="lns:loanApprovalLinkType"
   partnerRole="approver"/>
<partner name="assessor"
   serviceLinkType="lns:riskAssessmentLinkType"
   partnerRole="assessor"/>
</partners>

<variables>
  <variable name="request"
    messageType="lns:creditInformationMessage"/>
  <variable name="risk"
    messageType="lns:riskAssessmentMessage"/>
  <variable name="approval"
    messageType="lns:approvalMessage"/>
  <variable name="error"
    messageType="lns:errorMessage"/>
</variables>

<flow>
```
CHAPTER 3. BPEL

![Flow diagram of the Loan Approval Service.]

Figure 3.1: Flow diagram of the Loan Approval Service.

```xml
<links>
  <link name="receive-to-assess"/>
  <link name="receive-to-approval"/>
  <link name="approval-to-reply"/>
  <link name="assess-to-setMessage"/>
  <link name="setMessage-to-reply"/>
  <link name="assess-to-approval"/>
</links>

<receive name="receive1" partner="customer"
        portType="apns:loanApprovalPT"
        operation="approve" container="request"
        createInstance="yes">

  <source linkName="receive-to-assess"
          transitionCondition="bpws:getContainerData('request', 'amount') < 10000"/>
  <source linkName="receive-to-approval"
          transitionCondition="bpws:getContainerData('request', 'amount') >= 10000"/>
</receive>

<invoke name="invokeAssessor" partner="assessor"
         portType="asns:riskAssessmentPT"
```
Finally, it is possible to declare fault handlers. A handler will be executed if the corresponding fault is triggered by any of the process activity. In this case, if a loanProcessFault is raised, the process will reply to the customer with an invalidRequest fault (which is defined in the WSDL document), containing the error message.
3.2 Overview of BPEL Language

3.2.1 Process

The process element is the root of the BPEL document. It is composed of the following optional children: partnerLinks, partners, variables, correlationSets, faultHandlers, compensationHandlers, eventHandlers.

These elements permits the global declaration of the corresponding constructs (partners, variables, fault handler, ...). Then the process element contains the actual workflow definition with the top level activity declaration, one of the type described in the next section.

An activity can be a basic activity: receive, reply, invoke, assign, throw, terminate, wait, empty

or a structured activity which itself contains other activities: sequence, switch, while, pick, flow, scope, compensate

We briefly describes each of these activities in Section 3.2.4 and Section 3.2.5 but first starts with the variables definition and the expressions.
3.2.2 Variables

The variables element of the BPEL document, a direct child of the process element, provides the means to define the variables that will store the state of the interaction during the whole business process.

```xml
<variables>
  <variable name="ncname" messageType="qname"?
              type="ncname"? element="ncname"?/>
+</variables>
```

The type of each variable can be a WSDL messageType, an XML Schema simple type or an XML Schema element. The corresponding attribute should be used (messageType, type or element respectively).

Variables declared with a WSDL messageType are the one that can be used as an input, output or fault variable of the invoke, reply, receive, throw constructs. The variables are uninitialized at the beginning of the process, the assign and receive are the constructs that can change the value of variables.

3.2.3 Expressions

Four types of expression are found in BPEL document. Boolean expressions are found in conditions (join, transition, while and case conditions). Deadline-valued and duration-valued expressions are found in onAlarm and wait constructs. Finally general expressions are found in assignment for simple calculation in the logic of the process.

BPEL provides a flexible mechanism that enable to use any expression language for the expressions. However compliant implementations must support XPath 1.0 [29]. The specification explicitly states that XPath is not the ultimate solution and that the specification should evolve to support a more suitable expression language, including support for more nuanced constraints (e.g. string manipulation).

The expressions are built upon the variables of the process and its links status. Accessing the variables and the link status (see Section 3.2.6) is done through extension functions to XPath: bpws:getVariableProperty('variableName', 'propertyName') returns the value of the variable.
bpws:getLinkStatus('linkName') returns a boolean corresponding to the status of the link.

As example we have two boolean functions, one general expression (decreasing the counter 'tryCount' of the variable 'assessment'), one duration (3 days and 10 hours) and one deadline expression:
• bpws:getVariableProperty('stockResult', 'level')>0
• bpws:getLinkStatus('AtoB') and bpws:getLinkStatus('CtoB')
• bpws:getVariableProperty('assessment', 'tryCount') - 1
• 'P3DT10H'
• '2002-12-24T18:00+01:00'

3.2.4 Basic Activities

The receive activity

```xml
<receive partnerLink="ncname" portType="qname"
operation="ncname" variable="ncname"/>
</receive>
```

The receive activity defines a blocking wait for a message matching the portType and the operation defined in the construct. Actually the receive construct is one (with the reply construct) defining the external interface of the Web Service defined by the BPEL document, i.e. every receive element defines an operation available at this Web Service.

A business process MUST never have two receive activities for the same partner link, port type and operation active at the same time. In such a case the BPEL specification let the semantic undefined.

The reply activity

```xml
<reply partnerLink="ncname" portType="qname"
operation="ncname" variable="ncname" faultName="ncname"/>
</reply>
```

The reply activity allows the process to send a message in response to a message received through a preceding receive activity with the same partner link, port type and operation. If there is not a preceding matching receive activity, then the BPEL specification let the semantic of the process undefined.

Replies can be of two types. First, a reply can be a normal response, the variable attribute can be present and must match the declared response message type of the operation. The second type of response is a fault, in that case the faultName attribute is present and the variable attribute must match the message type of this fault’s variable.
The invoke activity

```xml
<invoke partnerLink="ncname" portType="qname"
    operation="ncname" inputVariable="ncname"
    outputVariable="ncname" >
    <catch faultName="qname" faultVariable="ncname"/>
    </catch>
</invoke>
```

The invoke construct permits defining the invocation of a partner’s Web Service. The invoke activity requires the partner link, port type and the operation and optionally an input variable and an output variable, the latter only in the case the interaction is synchronous (indeed, in the case of asynchronous interaction there is no direct response).

The nested catch construct defines the treatment of possible faults raised by the partner in handling the request. This can only occur in the case of synchronous interaction. The fault name must be one of the fault defined for this operation in the partner WSDL definition. A general catchAll element can also be used to treat all faults indiscriminately.

Finally a compensation handler can be defined. Compensation handler are aimed at rolling back the effect of the activity in case of failure. They can be invoked explicitly or by the default compensation mechanism. Compensation handler and their mechanisms are explained latter on.

The throw activity

```xml
<throw faultName="qname" faultVariable="ncname"/>
```

The throw constructs is aimed at raising internal fault. It can optionally specify additional data that will be available to the fault handler, e.g. to handle the fault or populate the messages sent to calling services.

The wait activity

```xml
<wait for="duration–expr"/>
```

or

```xml
<wait until="deadline–expr"/>
```
The wait construct allows to specify a delay or a deadline during/before which the process is blocked.

The empty activity

```xml
<empty/>
```

The empty activity is used when there is a need of doing nothing.

The assign activity

```xml
<assign>
  <copy> +
    <from>...</from>
    <to>...</to>
  </copy>
</assign>
```

The assign construct enables to copy messages from one variable to another. The variables are the one declared in the top variable tag. More than just copy, the assign construct permits the use of general expressions (see Section 3.2.3) to perform simple computation often required to define a business process, and assigning the result to a variable.

### 3.2.5 Structured Activities

The sequence activity

```xml
<sequence>
  [Activity] +
</sequence>
```

A sequence construct contains a list of activities, basic or structured, that must be executed sequentially, following the order in which they are listed. The sequence activity completes when the last activity of the sequence has completed.

The switch activity

```xml
<switch>
  <case condition="bool–expr"> +
    [Activity]
  </case>
  <otherwise>
```
The `switch` construct permits to define different behaviors of the process depending on boolean condition. The first activity whose condition is true will be executed. If all conditions are false the activity in the otherwise construct is executed, or an empty activity if the otherwise is not present. The boolean conditions are defined in Section 3.2.3.

**The while activity**

```
<while condition” bool–expr” >
   <Activity>
   </Activity>
</while>
```

The `while` activity will execute the inner activity, basic or structured, as long as the boolean condition is true.

**The pick activity**

```
<pick>
   <onMessage partnerLink=”ncname” portType=”qname” operation=”ncname” variable=”ncname”> +
      <Activity>
      </Activity>
   </onMessage>
   <onAlert for=”duration–expr”> *
      <Activity>
      </Activity>
   </onAlert>
</pick>
```

The `pick` activity is waiting for events to happen. To each event is associated an activity. The first occurring event will trigger the associated activity. All the subsequent event will be ignored. The `pick` activity completes when the triggered task completes.

There are basically two types of events. The first type is the arrival of a message, i.e. a partner invokes an operation, defined with a `onMessage` construct, similar to a `receive` activity. The second kind of event is alarm based. It can be defined, as in the case of the `wait` activity, with a duration or an absolute deadline. It is defined with the `onAlert` element.

**The flow activity**
The flow construct permits launching several activities concurrently. The links defined in the flow scope permits to enforce precedence between these activities, i.e. it permits synchronization. We present the link semantics in Section 3.2.6. The flow activity completes when all activities in the flow have completed. Completion of an activity however includes the possibility of it to be skipped. This happens when its enabling condition is false. This is explained in the Dead-Path-Elimination Section (3.2.6).

The scope activity

The scope constructs defines the context, i.e. local variables, fault handlers, compensation handlers and others, of the enclosed activities. Its structure is similar to the process constructs.

The compensate activity

Compensation handlers permit recovering from a fault on a case by case basis, its role is to recover to the state before the activity started. The compensation handler will be called with the following rule: 'Run all available compensation handlers for immediately enclosed scopes in the reverse order of completion of the corresponding scopes'. However a compensation handler can also be called explicitly through the compensate constructs.
3.2.6 Links

The `link` construct enables the expression of synchronization between concurrent activities, that is somewhere in the scope of a `flow` element. A link has a name, a source and a target. In its simplest expression, the link expresses the fact that the target activity waits for the source activity to complete.

Links are declared as children of the `flow` construct (see Section 3.2.5). Any activity can be declared source or target of a link, via the source or target attribute respectively. Each link must have exactly one source and one target.

The source activity may also declare a transition condition through the `transitionCondition` attribute. This condition when evaluated, just after the source activity completes, will determine the status of the link: positive or negative if the condition evaluates to true or false respectively. The status of the link will determine if the target activity is executed or skipped. The target activity can also specify a join condition, i.e. boolean function dependent on the incoming links’ status, that will define if the activity will be executed or skipped.

To determine if a concurrent activity can start or must be skipped three steps are taken. First it should be checked that without join condition the activity can start. Next, the status of all its incoming link should be determined (the source activities must have completed or been skipped). Finally, the activity is executed if the join condition evaluates to true and skipped otherwise.

In case of normal execution the transition condition sets the status of the outgoing links, or if absent, they are set to positive. however, if an activity is skipped, because its join condition evaluates to false, then all its outgoing links’ status are set to negative. The idea is that links set to false will lead to other activities being skipped. This phenomenon is called Dead-Path-Elimination.

Example

```xml
<flow suppressJoinFailure="yes">
  <links>
    <link name="buyToSettle"/>
    <link name="sellToSettle"/>
    <link name="toBuyConfirm"/>
    <link name="toSellConfirm"/>
  </links>
  <receive name="getBuyerInformation">
    <source linkName="buyToSettle"/>
  </receive>
</flow>
```
3.3 Conclusion

In this chapter, we presented the main concepts of the BPEL language. We saw that BPEL has four main characteristics. Firstly, BPEL has built-in constructs to interact with external WSDL Web Services through the `invoke`, `receive`, `reply` and `throw` constructs. Secondly, all data and messages are encoded in XML documents with a strong incentive to use XML Schema as the data type definition system. Thirdly, the data manipulation is done through XPath expressions. Finally, the `flow` construct associated with the `link` concept enables out-of-the-box definitions of concurrent executions.
Chapter 4

Model Checking BPEL Services

Contents

4.1 Petri Nets model .................................. 38
  4.1.1 Mapping BPEL into Petri Nets .............. 38
  4.1.2 Static Analysis of BPEL Processes ........... 43

4.2 Guarded Automata Model ......................... 45
  4.2.1 Guarded Automata Peers Model ............... 46
  4.2.2 LTL Properties .............................. 47
  4.2.3 From BPEL to Guarded Automata to Promela ... 48
  4.2.4 Results .................................... 52

4.3 Conclusions ..................................... 53

Model checking of BPEL services has been the attention of quite a lot of works recently.

In [3, 8] the authors use finite state machine models to formalize and check LTL properties of BPEL processes. Their modeling stays at the control flow level, abstracting the data manipulation part of the process.

Petri nets modelization is popular too. Indeed the control flow constructs found in BPEL are inspired by the concepts found in workflow definition languages and thus very close to them. The Petri net model has been shown to be a powerful tool to analyze workflow [23]. The Petri net based analysis of BPEL processes has been studied thoroughly in [15, 21, 31].

Abstract state machine models are also applied to the verification of BPEL processes in [5, 6].

Finally process algebra is also a natural fit to verify temporal logic properties as well as concurrent ones. Koshkina [16] verifies a simplified version of
BPEL, called the BPE-calculus, using the Concurrency Workbench of New Century (CWB), a general tool for verifying finite-state concurrent systems. Some efforts is also invested in process algebra in [7].

We choose to briefly introduce the Petri net modelization of van der Aalst et al. [18] since it is the first full formalization of control flow of BPEL processes. Moreover their formalization culminates in an automated tool that permits useful static verification.

Finally we present, also very briefly, the work of Bultan et al. Their work [9] is the first attempt to verify BPEL processes including modelization of the data manipulation part of the process, i.e. the XPath expressions.

4.1 Petri Nets model

W. van der Aalst et al. validate the soundness of using Petri nets as a model for the analysis of the control flow of BPEL processes. This is achieved through a complete mapping of the control flow constructs onto Petri nets.

As a first result, a fully complete formal semantics for BPEL processes at the control flow level is obtained [18]. This formal

Secondly, a tool, wofBPEL [19], automating this mapping and the static analysis of the resulting Petri net was developed. The automatic translation from BPEL to PNML, Petri net modeling language, is performed via BPEL2PNML. So that wofBPEL can statically analyze the corresponding Petri net and thus the modeled process.

We briefly presents this mapping now.

4.1.1 Mapping BPEL into Petri Nets

Activity

Activities are modeled as shown in Figure 4.1. An activity can have a number of targets (incoming links). For each of these targets we have two places, one for the positive status (true target) and one for the negative status of the link (false target). The join condition (see Section 3.2.6) is a boolean condition over the set of targets, it is modeled with the arcs linking the target and the 'sf' and 'st' transitions. There will be one 'st' transition for each set of target values such that the join condition evaluates to true. And exactly one 'sf' for each set of target values such that the join condition evaluates to false.

When the activity has completed the transition 'et' or 'ef', depending if the activity has been skipped or not, forwards the token to the correct value of the sources and puts a token in the end place.
The activity itself is a sub-petri-net, one of the constructs presented below. The source and target places are components of links that we present now.

**Links**

![Petri net representing a Link](image)

Figure 4.2: Petri net representing a Link [25].

The links (see Figure 4.2), in the model of van der Aalst are actually a model of the transition condition. If the activity has been skipped, the false source has a token, this token is passed along to the negative target (death-path-elimination). If the previous activity executed normally, the positive source has a token, this token is passed along to the negative target (death-status).

For every source and target of an activity we will have such a construct modeling its link. However some activities don’t have any source, nor a target, these are simply modeled as below.
CHAPTER 4. MODEL CHECKING BPEL SERVICES

Sequence

![Petri net representing a Sequence](image)

Figure 4.3: Petri net representing a Sequence [25].

A Sequence construct is trivially modeled by putting the activities in sequence linking each activity to the next (Figure 4.3. The Sequence completes when the last activity has completed.

Switch

A switch between Activities A, B and C can be modeled as in Figure 4.4. With one token in the started place, one of the Activities A, B or C will execute. The actual switch condition is not modeled and thus is part of the 'black-box' assumptions of the model, it is supposed that for any of the condition there is some run in which this condition will be true.

Moreover in the BPEL specification a switch statement is an ordered sequence of activities with an associated condition, the first activity with its condition evaluating to true will be executed. This model, since it’s not taking into account the actual condition, cannot detect that an activity can never execute if, for example, its associated condition is true only when a prior activity’s condition is true. Finally the BPEL specification asks for an empty activity being executed when none of the condition is true. How the translation process deals with this problem is not clear (adding an empty activity or not?).

While

The while construct will execute Activity A as long as a condition is true (Figure 4.5). This model abstracts the condition and thus suppose Activity A will or will not execute eventually. The While activity completes as soon as the condition is false.
3 show how such links are mapped onto Petri net fragments, which extend the Petri net fragment resulting from Figure 3.8. Figure 7 shows how we can map this onto a Petri net fragment. If the associated condition evaluates to "true", then transition ct (condition true) will enable the outgoing activities. In the hierarchical activity structure, all contained activities are enabled in the same order as they occur. In the while activity, all activities are enabled whenever the while activity occurs, and the transition ct (condition true) is enabled as long as the condition evaluates to "true".

Figure 4.4: Petri net representing a Switch [25].

Figure 4.5: Petri net representing a While [25].
This is achieved by the 'aj' transition that can fire only when all the sub-activities have completed. The Flow completes when all the concurrent sub-activities have completed.

### Pick

A pick construct contains a set of activities each waiting for an event to start (Figure 4.6). The first activity that received the expected event will execute. The pick activity completes when the started inner activity has completed.

### Flow

In a flow construct, all activities are executed concurrently, this is modeled by producing a token in the started place of each of the activities (Figure 4.7). The Flow completes when all the concurrent sub-activities have completed. This is achieved by the 'aj' transition that can fire only when all the sub-activities have completed, placing a token in the corresponding place.
Basic Activity

A Basic Activity (reply, receive, invoke...) is modeled with the Petri net in Figure 4.8. The 'c' transition denotes the normal execution of the activity, the 'a' transition models its abortion and the 'b' transition is for rolling back of the activity.

4.1.2 Static Analysis of BPEL Processes

Woflan [24] is a state-of-the-art Petri-net-based analysis tool for verifying the correctness of a workflow process. Woflan was developed by Eric Verbeek and Wil van der Aalst as a tool to detect errors in workflows formalized as WF-net. They extended this tool in a natural way to handle BPEL processes by simply plugining-in an automatic translator of BPEL processes to PNML, the Petri-net modeling language.

Reachability Analysis

Reachability of all activities is desired for any BPEL process, i.e. the possibility for any one of the activities that appear in the process definition to be executed. This won’t be the case if we have some dead transition in the corresponding Petri net.

The authors deployed, without receiving any warning from the BPEL container, the BPEL process described below. It is easy to see that the empty activity named ‘nonEric’ can never execute. Indeed, for this activity to execute the link ‘linkEric’ must be activated and thus the condition of the first ‘case’ of the switch construct must be true. This is not possible since we are already in the otherwise condition. The process will wait infinitely for the link to be activated.

![Figure 4.8: Basic activity mapping [25].](image)
CHAPTER 4. MODEL CHECKING BPEL SERVICES

Using WofBPEL the deadlock was reported. Moreover the tool accurately points the empty task in the otherwise construct to be the cause of the error.

Competing Message-Consuming Activities

In Section 3.2.4 we mention that “A business process MUST never have two receive activities for the same partnerLink, portType and operation active at the same time”. In short, a process cannot have two competing message-consuming activities, competing being interpreted as above: same partner link, port type and operation. More generally, this constraint must hold for any consuming activity, which are not only the receive activities but also the pick activities and the event handlers.
In [20] the petri nets model permits detection of processes violating this constraint. In the example above, two sequences are run concurrently. If invoke 'A1' and receive 'rcv2' have completed, then 'rcv3' and 'rcv1' are competing for the message 'pl2/pt2/op2'. To discover this competing message-consumption violation a full state space search was needed.

Garbage Collection of Queued Messages

Using the same model, it is possible to compute for each activity 'a' of a process all the possible message types that could eventually be consumed by the process after the execution of 'a'. Computing this set for each activity enables the annotation of the BPEL process, by indicating for each activity the message types that are still relevant to the process. If the process, after execution of activity 'a', receives a message that is not in the corresponding set of message types of 'a', then this message can be discarded (garbage collected).

4.2 Guarded Automata Model

In the Loan Approval Service, we find 3 partners: the Loan Approval Service, the assessor and the customer. Each of these services could possibly be implemented as a BPEL process. They can be viewed as interacting distributed processes (Figure 4.9).

In this paper [9], Xiang Fu et al. expose a method to transform interacting BPEL processes into Promela, the input language of the well-known SPIN model checker. This enables them to check the BPEL processes against LTL properties.

Moreover, building on previous work [10], their translation allows for the handling of XPath expressions which are central to control the flow of the BPEL process. This constitutes an interesting enhancement over older approaches that were completely abstracting the data manipulation part of the process. This leads to a more fine grained verification.
Finally they exhibit the synchronizability concept as a way to handle infinite queues. This important result permits the complete verification of properties of a large class of Web services with unbounded input queues.

### 4.2.1 Guarded Automata Peers Model

As intermediate representation between BPEL and Promela the authors use a model based on guarded automata communicating through message queues. The model consists of three elements (Figure 4.10). Firstly a conversation schema declares the communicating peers, i.e. the BPEL processes involved, and all the possible type of messages that they can exchange. In the example below there are three peers (A, B and C) with msg1 the only message type that A can send to B, msg2 and msg6 the possible types of message peer B can send to A, and so on. Secondly, there are the guarded automata peers that represent the BPEL processes them self. A message queue that store the incoming messages sent by the other peers is attached to each automata. Finally a virtual watcher records all the messages by all peers in the order that they are produced. It actually records the whole conversation.

A guarded automata, here, is a tuple \((M_i^{in}, M_i^{out}, L_i, s_i, F_i, \Delta_i)\). \(M_i^{in}\) is the set of incoming message types, \(M_i^{out}\) is the set of outgoing message types, \(L_i\) are the local variables, and \(T_i, s_i, F_i\) are the states, the initial state and the set of final states. All the messages, in and out, as well as all the values of the local variables are all XML documents. \(\Delta_i\) is the transition relation.

A transition \(t \in \Delta_i\) is in one of the following three forms: \((q_1, g, q_2)\), called a local transition, \((q_1, ?a, q_2)\), a receive transition and \((q_1, (b, g), q_2)\), called a...
send transition. Each of these transitions change the state of the peer from $q_1$ to $q_2$. The guard $g$ is a boolean condition and a set of assignments. Send and local transition are executed only if the guard condition evaluates to true. The receive transition executes when the corresponding message is at the front of the message queue of the peer. The assignments of send transitions represent the content of the message being sent. The assignment of local transitions update the local variables. The guard conditions and the guard assignments are XPath expressions, just as in the BPEL process.

In order to verify properties on the conversation, i.e. the sequence of messages exchanged, all the messages sent between peers are recorded in a virtual watcher, concept introduced in [11]. Every message being added to a peer’s message queue is also added to the virtual watcher in the order they are produced.

### 4.2.2 LTL Properties

The authors used LTL formulas [17] to express the properties on the conversations. The LTL formulas are based on two type of atomic propositions, a message type $m$ or a predicate on the message type $m.pred$.

**Example**

- $G($register $\rightarrow F($accept $\lor$ $reject)$)
- $G($register$/$stockID==a $\rightarrow F($request$/$stockID==a)$)
The first proposition (which has nothing to do with the previous example), based on message types as atomic propositions, states that each message 'register' will be eventually followed by an 'accept' or a 'reject' message. The second, an example of formula with predicates on message content, states that every stockID that appears in a register will finally appear in a request message.

In order to verify that a BPEL process satisfies a property, the author developed a translation tool from BPEL to the guarded automata peers model and from this model to Promela, the input language of the SPIN model-checker which verifies LTL properties.

However even in a simple case verifying properties is not decidable [9]: “Given a composite web service S with finite content and an LTL property ϕ, checking that S satisfies ϕ is undecidable”. A web service composition is said to be with finite content when the set of all possible messages (not just the message types) in the system is finite. Since this result is due to the presence of unbounded message queues, putting a bound on these queues enables verification and the use of the SPIN model checker.

### 4.2.3 From BPEL to Guarded Automata to Promela

#### BPEL to Automata Model

Translating the BPEL definition to the guarded automata can be done in two steps. First, the conversation schema must be derived. Then each guarded automaton is produced from the BPEL document.

The conversation schema consists of, firstly, the declaration of the peers, one per BPEL input document. Secondly the message types, as well as their senders and receivers, are easily deduced from the BPEL and WSDL documents of the partners in all interactions. These message types of the conversation schema must be unique for a sender, a receiver and an operation.

The translation of BPEL documents into extended guarded automata is similar to its translation into Petri nets as in the previous section. The table below exemplifies the translation of the assign, receive, invoke, sequence and flow constructs. The XPath expressions are directly translated into guard conditions and assignment expressions.

The flow construct is translated through a Cartesian product of all sub-activities (interleaving). This may lead to a great number of states in the resulting automaton and has a scaling problem as a consequence.
### BPEL 
Sample Code 
Translation

<table>
<thead>
<tr>
<th>BPEL</th>
<th>Sample Code</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>assign</td>
<td><code>&lt;assign ...&gt;</code>&lt;br&gt;<code>&lt;copy&gt;</code>&lt;br&gt;<code>&lt;from=&quot;yes&quot;/&gt;</code>&lt;br&gt;<code>&lt;to var=&quot;aprvInfo&quot; part=&quot;accept&quot;/&gt;</code>&lt;br&gt;<code>&lt;/copy&gt;</code>&lt;br&gt;<code>&lt;/assign&gt;</code></td>
<td><img src="image" alt="assign translation" /></td>
</tr>
<tr>
<td>receive</td>
<td><code>&lt;receive ...</code>&lt;br&gt;<code>operation=&quot;approve&quot;</code>&lt;br&gt;<code>variable=&quot;request&quot;</code></td>
<td><img src="image" alt="receive translation" /></td>
</tr>
<tr>
<td>invoke</td>
<td><code>&lt;scope&gt;</code>&lt;br&gt;<code>&lt;invoke ...</code>&lt;br&gt;<code>operation=&quot;approve&quot;</code>&lt;br&gt;<code>invar=&quot;request&quot;</code>&lt;br&gt;<code>outvar=&quot;aprvInfo&quot;/&gt;</code>&lt;br&gt;<code>&lt;catch ...</code>&lt;br&gt;<code>faultname=&quot;loanfault&quot;</code>&lt;br&gt;<code>&lt; ... handler1 ... /&gt;</code>&lt;br&gt;<code>&lt;/catch&gt;</code>&lt;br&gt;<code>&lt;/scope&gt;</code></td>
<td><img src="image" alt="invoke translation" /></td>
</tr>
<tr>
<td>sequence</td>
<td><code>&lt;sequence ...&gt;</code>&lt;br&gt;<code>&lt; ... act1 ...&gt;</code>&lt;br&gt;<code>&lt; ... act2 ...&gt;</code>&lt;br&gt;<code>&lt;/sequence&gt;</code></td>
<td><img src="image" alt="sequence translation" /></td>
</tr>
<tr>
<td>flow</td>
<td><code>&lt;flow ...&gt;</code>&lt;br&gt;<code>&lt; ... act1 ...&gt;</code>&lt;br&gt;<code>&lt;source</code>&lt;br&gt;<code>linkname=&quot;link1&quot;</code>&lt;br&gt;<code>condition=&quot;cond1&quot;/&gt;</code>&lt;br&gt;<code>&lt;/act1&gt;</code>&lt;br&gt;<code>&lt; ... act2 ...&gt;</code>&lt;br&gt;<code>&lt;target</code>&lt;br&gt;<code>linkname=&quot;link1&quot;</code>&lt;br&gt;<code>&lt;/act2&gt;</code>&lt;br&gt;<code>&lt;/flow&gt;</code></td>
<td><img src="image" alt="flow translation" /></td>
</tr>
</tbody>
</table>

Figure 4.11: Translation to Guarded Automata Model [10].
Automata to Promela

Translating a model composed of a conversation schema and the corresponding guarded automata into Promela is not an easy task. The main difficulties reside in the translation of the types and of the XPath expressions. This translation can be presented in 4 steps: types, processes, channels and XPath.

Types  The types of the variables in a Promela specification are essential to the success of the verification. Indeed, the state space explosion problem is directly linked to the size of the domain of the system variables, i.e. their types.

Each type of the specification is mapped to a typedef in Promela. Strings are used only as constants, they are mapped to the enumerated type of Promela, \texttt{mtype}. Boolean and numeric types are mapped to the corresponding Promela types.

Processes  Each automaton is translated into a Promela process, \texttt{proctype}. Inside each proctype, corresponding to an automaton, i.e. corresponding to a BPEL document, we find the declaration of the local variables, corresponding to each variable of the BPEL document. Moreover an additional variable is declared to record the current state of the automaton.

The process itself is composed of a single loop. This loop contains one big switch/case statement. Each branch, or case, of this statement corresponds to a transition of the automata. The condition for the branch to be taken is, firstly that the current state of the automaton is the source state of that transition and secondly that the corresponding guard condition evaluates to true. The guard condition is an XPath expression that must be translated as explained below. Furthermore, for receive transitions, the corresponding message is also required, i.e. a message of that type must be in the process channel. A send transition is translated in a similar way. Finally a final transition will be translated into a jump out of the loop.

Channels  The queue of the automaton is associated with a Promela communication channel. The Promela channels have a finite fixed length, this corresponds to bounded queues. While this is a serious restriction on the model, the author exhibited a property 'synchronizability' that permits the full model checking of a large class of BPEL processes with unbounded queues.
**XPath** XPath translation is hard. In [10], the author proposes a method for translating bounded XML data and XPath expressions into Promela. Since SPIN is a finite state system verification tool, the model variables must all be bounded. Thus the XML data in this case are supposed to be bounded, i.e. there should only be a finite number of XML documents as possible value of a variable.

While we will not introduce the method here, we give an example of the translation that speaks for itself on the complexity of the task. We observe that this simple XPath leads to a complex Promela translation containing 5 integer and 4 boolean variables. This could leads to into scalability issues due to the state explosion problem.

```plaintext
$\text{request} // \text{stockID/int}() = \\
$\text{register} // \text{stockID[int}>5][\text{position}=\text{last}]/\text{int}() \\
```

translates into:

```plaintext
/* result of the XPath expression */
bool bResult = false;
/* results of the predicates 1, 2, and 1 resp. */
bool bRes1, bRes2, bRes3;
/* index, position(), last(), index, position() */
int i1, i2, i3, i4, i5;

i2 = 1;
/* pre-calculate the value of last(), store in i3 */
i4 = 0; i5 = 1; i3 = 0;
do :
    :: i4 < v_register.requestList.stockID.occ
        ->
    /* compute first predicate */
        bRes3 = false;
        if :
            :: v_register.requestList.stockID[i4].intvalue > 5
                -> bRes3 = true
            :: else -> skip
        fi;
        if :
            :: bRes3 -> i5++; i3++; 
            :: else -> skip
        fi;
    i4++;
    :: else -> break;
od;
/* translation of the whole expression */
i1 = 0;
```
CHAPTER 4. MODEL CHECKING BPEL SERVICES

The full translation is shown in [9]. More important are the concrete results of the application of this technique and their scalability problems. This is presented in the next section.

4.2.4 Results

In contrast to previous work, Bultan et al. present an approach that does not abstract the data manipulation part of the BPEL process. Their tool, WSAT [22], permits the verification of LTL properties of interacting BPEL processes with bounded queues. WSAT also checks if the process verifies the
synchronizability property and if it is, proceed to the appropriate modification of the model that permits the verification of the LTL properties, the results are then still valid for unbounded queues.

While a class of processes can be verified with unbounded queues, the processes that are synchronizable, there are still lots of processes that do not verify that property and thus the verification of LTL property are limited to finite queues for these processes. Moreover, the method does not scale well. In their conclusion, the authors report on their experiments on a model containing 3 peers: “When integer domain is set to [0,1], the verification time is 3 seconds and memory consumption is around 50MB. When the domain is increased to [0,3], the memory consumption grows to over 600MB. However, our experience shows that SPIN is still useful in identifying errors in protocols by restricting the data domains”.

4.3 Conclusions

Verification of BPEL processes has been investigated for some times now. Two approaches have been subject to research.

The first approach makes abstraction of XPath expressions, the data manipulation part of a BPEL process, it concentrates fully on the control flow level. This first approach is quite successful and several automated tools have been developed enabling verification of soundness and LTL properties.

The second tries to include XPath expressions in the model. While some results are obtained, the method does not scale well, verification is limited to very simple processes.
Chapter 5

Testing BPEL Web Services

Contents

5.1 Test Suites ........................................ 54
5.2 Generating Test Cases using SPIN for BPEL Services ............................ 55
  5.2.1 BPEL to Promela ......................... 56
  5.2.2 SPIN to Test Cases ..................... 58
  5.2.3 Results .................................. 58

5.1 Test Suites

The use of model checking and formal methods helps reducing the number of errors in the early stages of software development. In the later stages, testing is a popular method for verifying the correctness of the implementation against its expected behavior.

Testers usually write a lot of test cases. A test case or unit test is the description of an input on which the software is to be executed. A test case can also contain the expected result or response from the program. The program passes the test if the actual response is the response that was expected. A set of test cases is called test set or test suite.

Judging the quality of a test suite is the subject of a lot of researches [32]. Coverage analysis is a popular way of assessing the quality of a test set. However there are several different coverage criteria that have been proposed and argued.

Statement coverage checks that every statement is executed.
CHAPTER 5. TESTING BPEL WEB SERVICES

Branch coverage checks that every branch is actually taken.

Path coverage check that every path is taken.

Mutation coverage checks that the tests discriminate against every small change in the code, i.e. mutation of the code.

Let consider the following code:

```java
if(t==0) {
    a = 1;
}
if(s==0} {
    b = 1;
}
```

To have 100 percent statement coverage it is sufficient to have \( t==0 \) and \( s==0 \), so a single test is required. To have full branch coverage the true and false branch of each test must be covered. It is sufficient to have two tests, one with \( t==0 \) and \( s==0 \) and another where \( t!=0 \) and \( s!=0 \). Finally to have complete path coverage at least four tests are required, one for each combination of possible values of \( t==0 \) and \( s==0 \).

The mutation coverage is not illustrated here, it strongly depends on the mutation operator (e.g. syntactic mutation of the code). To achieve full coverage the results of the tests must detect all non-equivalent mutants.

Getting a test set that is of good quality with regards to one of the criteria above can be tedious. Automatically generating these test cases alleviates that task.

5.2 Generating Test Cases using SPIN for BPEL Services

Model checking can be used to help generate test cases [14, 2]. Indeed, model checkers like SPIN or SMV check if a property, specified as an LTL formula, holds for a given model. If it does not hold, then the model checker extracts the trace of an execution on the model that violates the property, i.e. a counterexample. In order to get a test case for a property \( C \), it suffices to run the model checker on the model with the property 'C never holds'. The model checker will, thus, exhibit a counterexample that describes an execution where the \( C \) property holds. This counter example, if it exists, can be the source of a test case.
In [12], the authors used SPIN to generate test cases for BPEL processes. Firstly, they transform the BPEL process into Promela. Next, they identify all the transitions, from one activity to another, of the BPEL process. Finally the generation proceeds as follows. To each transition, X, is associated the LTL property: 'Transition X is never executed'. The SPIN model checker is run with the Promela model and each of the above LTL properties. The counterexample is an execution that triggers the transition X. A test case can be produced accordingly. The overall process is illustrated in the Figure 5.1.

This method will generate a test suite that covers all transitions of the BPEL process, provided that a counterexample can be found for each transition. Other coverage criteria can be used, so long as one can express it as an LTL formula.

5.2.1 BPEL to Promela

The translation into Promela is quite similar to the one presented in Section 4.2.3. However, instead of going through an intermediate representation, i.e. the guarded automata model, the authors present a direct transformation. While this Promela translation has a lot in common with the previous one, there are also important differences.

The BPEL process is translated into a Promela process construct, as are all the partner processes. However, unlike in the former translation, the partner processes are deduced from the BPEL document, i.e. no need here to provide (simple) BPEL documents for each partner. These partners are deduced from the partner definition in the BPEL document, via the associated WSDL definition. Additionally, the behavior of the partner, the messages it sends, is deduced by analysis of how the process actually uses the information.

For example, in the Loan Approval Service, the process receives a riskAssessment.risk from the assessor. The value of the risk variable is tested against the constant 'low'. Therefore the corresponding Promela process will send 'low' or 'other' as risk response in order to cover all branches of the test.

The behavior of the deduced Promela process follows 3 rules:

1. If the BPEL document has no reference to the data, then its unique possible value is undefined
2. If the BPEL document compares it to a numerical constant: the possible value are the constant as well as a lower and an higher value
Figure 5.1: Generation of test cases with SPIN [12].
3. If the value is discrete in the BPEL document: the possible values are all the discrete values as well as other.

Afterwards, the partner processes and the process in itself are mapped into the Promela specification. The processes port types are transformed into Promela channels, the message types into Promela typedefs and each data type is discretized from the BPEL type into boolean for all data, except for boolean that are mapped to Promela boolean. The invoke and receive activities are modeled through the ! and ? channel operators. The flow controls, i.e. sequence, while, link, flow constructs..., are then mapped (manually?) into the Promela specification.

5.2.2 SPIN to Test Cases

To fulfill the transition coverage criteria, LTL formulae are used to produce counterexamples triggering the targeted transitions. In their implementation the authors use a boolean variable for each transition, e.g. for transition X the variable is transX. This variable will be set to true in the Promela model when the transition is executed. The corresponding LTL formula will thus be: \( \text{\texttt{[] !transX}} \) (i.e. always not transX).

To deduce the actual test case from a counterexample, one must simply use the sequence of exchanged messages. Indeed, they give a complete description of the interaction between the partners leading to the execution of the targeted transaction.

To produce the complete transition coverage, one has to run the model checker with the LTL formula of not yet executed transitions as long as such transition exists. In some cases some transitions are just unreachable, however the authors do not explicitly treat this case.

5.2.3 Results

The authors applied their technique to the Loan Approval Services. In less than three runs they obtained a test suite covering all transitions. The performance on this simple example was really good.

The paper is only preliminary results and it is the first and only one investigating the application of model for test suite generation for BPEL processes. The translation process is not automated. Moreover, the example investigated is very basic, it has only trivial XPath expressions. The application to BPEL process with more complex XPath must be studied. Even more, as we saw in Section 4.2.3, a generic translation of BPEL to Promela, including XPath expression, has a scalability problem. The scalability of the
present technique, i.e. with more complex, real life BPEL processes, must be looked into.
Chapter 6
Conclusions and Future Works

BPEL is a fully fledged, high level programming language for business processes based on Web Services. As with any other software system, the need to verify and validate BPEL processes is clear. Commonly two methods are used for verifying and validating softwares: model checking and testing.

Model checking of BPEL processes has been closely investigated. These researches about it split into two approaches.

The first approach makes abstraction of the XPath expressions, the data manipulation part of a BPEL process, it concentrates fully on the control flow level. This first approach is quite successful and several automated tools have been developed enabling verification of soundness and LTL properties.

The second approach tries to include the XPath expressions in its model. While some results are obtained, the method does not scale well, verification is limited to very simple processes.

On the testing side of the validation, very few works explicitly tackle the generation of test suites based on model checking techniques. The one we presented in Chapter 5 is a proof of concept. More investigations are needed to assess the scalability of the technique.

Abstract interpretation techniques like the one used in the Blast model checker [1] is an alternative model checking technique. Its application to BPEL processes and especially to XPath expressions could yield interesting results and should be explored. Moreover Blast was extended to automatically generate test suites from counterexamples [4]. However, more research is needed to handle with Blast the concurrency capability built into BPEL.
List of Figures

1.2 The Loan Approval Service. ..................................... 8
2.1 XML Schema data types [30] ............................... 13
3.1 Flow diagram of the Loan Approval Service... .............. 26
4.1 Petri net representing an activity. .............................. 39
4.2 Petri net representing a Link [25]. ............................. 39
4.3 Petri net representing a Sequence [25]. ...................... 40
4.4 Petri net representing a Switch [25]. .......................... 41
4.5 Petri net representing a While [25]. ........................... 41
4.6 Petri net representing a Pick [25]. ............................. 42
4.7 Petri net representing a Flow [25]. ............................ 42
4.8 Basic activity mapping [25] .................................... 43
4.9 Loan Approval Service Interactions [9] ......................... 46
4.10 Guarded Automata model [10] ................................. 47
4.11 Translation to Guarded Automata Model [10] ................. 49
5.1 Generation of test cases with SPIN [12] ..................... 57
Bibliography


