EXTENSION OF THE BUSINESS PROCESS EXECUTION LANGUAGE TO SUPPORT COMPLEX CONTROL FLOW PATTERNS

JEAN-PAUL AINAM

EXTENSION OF THE BUSINESS PROCESS EXECUTION LANGUAGE TO SUPPORT COMPLEX CONTROL FLOW PATTERNS

Jean-Paul Ainam
PG/12/0579
BSC in Computer Maintenance and Software Engineering

BEING A PROJECT SUBMITTED IN THE DEPARTMENT OF COMPUTER SCIENCE, SCHOOL OF COMPUTING AND ENGINEERING SCIENCE, IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE AWARD OF THE DEGREE OF MASTER OF SCIENCE, BABCOCK UNIVERSITY, ILISHAN-REMO, OGUN STATE, NIGERIA

2014

CERTIFICATION

This Dissertation titled, EXTENSION OF THE BUSINESS PROCESS EXECUTION LANGUAGE TO SUPPORT COMPLEX CONTROL FLOW PATTERNS, prepared and submitted by JEAN-PAUL AINAM in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE (Computer Science), is hereby accepted

____________________________________ (Signature and Date)

OKOLIE S.O, PhD.
Supervisor
(Associate Professor)

Accepted as partial fulfillment of the requirements for the degree of MASTER OF SCIENCE (Computer Science)

____

Prof. Ademola S. Tayo
Dean, School of Postgraduate Studies

DEDICATION

To the two pillars of my life God and my parents

ACKNOWLEDGEMENTS

First, I would like to express my sincere gratitude to God for the energy, strength and intelligence He gave me to do this academic work. My appreciation goes to my supervisor Dr. Okolie S.O for his support throughout this academic project. Through his guidance he successfully helped me to achieve this Master Project. My gratitude also goes to Babcock University and Adventist University Cosendai, for the scholarship I received from them. My deep appreciation goes to all my lecturers for the training I received from them. More especially I want to thank Mrs Kalejaiye A.S from the English Department who did not spare any effort in helping to improve my proficiency in English. My gratitude also goes to all my coursemates at Babcock School of Computing and Engineering Science. Last but not the least, I would like to thank my father Jacques Wang-Kel, my beloved mother Rebecca Maidjamon, my dear brothers and sister for their spiritual, moral and financial support throughout my life.

Abstract

The Business Process Execution Language (BPEL) has emerged as the de facto standard for Web Service composition and was created to enable the orchestration of web services. However investigations have shown that BPEL does not offer full support for some complex control flow patterns, namely, multi-merge, discriminator, arbitrary cycle, interleaved parallel routing and milestone patterns. Little research has been done in this area and existing solutions are solely based on work-around solutions rather than an extension. In this project, an approach to extend the BPEL standard is presented. This extension, called xBPEL uses the Java Architecture for XML Binding (JAXB) for handling the unmarshalling (reading) of XML instance documents into Java Objects, and marshalling (writing) Java Objects back into XML instance documents. The work has intended to extend BPEL to support a wide range of control flow patterns. In fact, xBPEL is built on top of the existing specification BPEL and offers various extension points for unsupported control flow patterns by defining a new set of constructs, attributes and rules. For each control flow pattern, a prototype's implementation showing the feasibility of the proposed extension is presented. At the end of this project, an architectural approach to integrate existing BPEL engines with the proposed extension was presented. Therefore, further works can explore how to extend BPEL to all other Workflow Patterns not yet supported. Another option is to improve the scalability and interoperability across BPEL engines by exploring how an instance of a process in execution can be transferred from one engine to the other without the need of redeploying the service.

Keywords: Business Process Execution Language (BPEL), Workflow Patterns, Control Flow Patterns, eXtensible Markup Language (XML)

Word Count: 262

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LISTINGS

CHAPTER ONE
INTRODUCTION

1.1. Background of the Study

The Service-Oriented Architecture (SOA) is an architectural approach using technology to present business processes as reusable services. Services are autonomous by nature and can be combined to build new services – an activity generally known as composition. There has been significant development in the technologies that enable creation of web services such as the Business Process Execution Language (BPEL) standard. The origins of BPEL go back to Web Service Flow Language (WSFL) and XLANG. It is serialized in XML and aims to enable programming in the large which generally refers to the high-level state transition interactions of a process. It includes information such as when to wait for messages, when to send messages, when to compensate for failed transaction, etc.

BPEL is indeed an XML-based language under consideration of the Organization for the Advancement of Structured Information Standards (OASIS). It has become the most popular workflow language accepted in industries (Ko, Lee, & E. W., 2009) and has proved to be sufficiently robust to handle most of the scientific workflow in direct or indirect applications (Akram, Meredith and Allan 2006; Slominski 2007). It is also one of the most widely adopted approaches to orchestrate services (Turner, 2005) and has been adopted industrial-wide as the de facto service composition standard (Jonathan, et al. 2009; Tim, et al. 2008; Kloppmann, et al. 2004). BPEL is in fact an orchestration language, and not a choreography language. The primary difference between orchestration and choreography is executability and control. An orchestration specifies an executable process that involves message exchanges with other systems, such that the message exchange sequences are controlled by the orchestration designer. Choreography specifies a protocol for peer-to-peer interactions, defining, for example the legal sequences of messages exchanged with the purpose of guaranteeing interoperability. Such a protocol is not directly executable, as it allows many different realizations. Yet, choreography can be realized by writing an orchestration for each peer involved in it. In simply terms, orchestration refers to the central control of behavior of a distributed system while choreography refers to a distributed system which operates according to rules but without centralized control.

BPEL provides several basic activities which enable interaction with the services being arranged in the workflow. A workflow consists of any sequence of actions that define the structural organization of partners and activities, or determine who processes what and how the processes are done (Hackmann, Haitjema, Gill, & Roman, 2006). At a higher level of abstraction, application architects use concepts such as workflow or dataflow to describe the movement of information as documents or messages through a set of processes. Designed to support the modelling of two types of processes (executable and abstract processes) BPEL standard attempts to meet these objectives by providing features to adequately represent the business logic of the process (Petia, Wil M.P., & Marlon, 2002).

In fact, many traditional business processes – such as loan approval, insurance claim processing, and expense authorization – can be modeled naturally as workflows. This has also motivated the development of Business Process Execution Language to describe these processes using a common language. Each task in a BPEL process is represented as a service that is invoked using the Simple Object Access Protocol (SOAP). However BPEL does not provide direct support for some of the workflow patterns existing in an organization.

For example, "*lay foundation"*, "*order material*" and *"book labourer*" are three tasks which can occur in parallel as separate process branches. After each of them completes, the *"quality review*" task is run before that branch of the process finishes. This type of workflow pattern, known as multi-merge, models the convergence of two or more branches into a single path (see section 2.1.3); that is, a point in the workflow process where, based on a decision or workflow control data, a number of branches are chosen. However, the current version of BPEL (WS-BPEL 2.0) does not provide any construct for such workflow pattern, making it hard for business architects to compose it. This illustrates one of the seven control flow patterns not supported by BPEL. The remaining control flows not supported by BPEL are described in section 2.1.3.

In sight of the above limitation of BPEL, a need to extend the language to support complex control flow patterns becomes necessary. The extension is all the more necessary as it would allow business architects to compose business processes more effectively. This work therefore covers control patterns not supported by BPEL and goes ahead to propose xBPEL, an extension of BPEL to support these control flow patterns.

The main advantage of this work is that there is no need to recreate a new set of languages or standards to support complex control flow patterns. Support is only achieved by extending the existing business language, BPEL. This approach also allows development tools such as Eclipse BPEL Designer and BPEL engines such as ActiveBPEL, ActiveOS and Oracle SOA Suite to reuse the proposed extension without any change on the design architecture of the applications software being used for development (tools and engines). The terminologies involved in this project have the following definition:

Definition 1.1: A **business process** consists of a set of one or more linked procedures and activities which collectively release a business policy goal, normally within the context of an organizational structure defining functional roles and relationships.

Definition 1.2: A **process** is a formal point-of-view on the business process, supporting automated manipulation such as modeling or enactment by a workflow management system. It is a set of parallel or serial activities and their relationships, criteria to indicate the beginning and the termination of a process and information about individual activities, such as participant, associated IT applications and data.

Definition 1.3: The **Business Process Execution Language** is an XML-based language and an orchestration language used to describe execution logic of Web Services applications by defining their control flows and providing a way for partner services to share a common context.

Definition 1.4: An **orchestration language** specifies an executable process that involves message exchanges with other systems, such that the message exchange sequences are controlled by the orchestration designer.

Definition 1.5: From the Web Service's point of view, a **partner** is simply the service provider that interacts or communiques with the process.

Definition 1.6: A **Web service** is a method of communications between two electronic devices over the World Wide Web. It is a software function over the web with the service always on as in the concept of utility.

Definition 1.7: An **activity** is a description for a work stage representing a logical step within a process.

Definition 1.8: An **instance of an activity** is the representation of an action during a single process execution.

Definition 1.9: The **Business Process Management Notation** is a standard which creates a standardized path to fulfill the gap between the analysis and the implementation of a business process.

Definition 1.10: An **event** describes what circumstances a function or a process works (starting event) or which state a function or process results in (ending event).

Definition 1.11: A **control flow** connects events with functions, process paths, or operators creating chronological sequence and logical interdependencies between them.

Definition 1.12: A **process path** shows the connection from or to other processes.

The rest of this chapter is organized as follow: Section 1.1 gives the motivation for the choice of this topic; section 1.2 defines the problem statement followed by the aim and objectives of the work in section 1.3. Section 1.4 and 1.5 end this chapter by providing scope and overall structure.

1.2. Motivation

The Service-Oriented Architecture (SOA) conceptualizes functionality as services. It is a concept and a strategy for using technologies to build business automation solutions and can be achieved through the use of web services composition like Business Process Execution Language (BPEL). BPEL is, indeed an OASIS (Organization for the

Advancement of Structured Information Standards) standard for modeling and executing business processes by orchestrating Web Services. Since the first version of BPEL published by the OASIS in 2003 (Tony, et al., 2003), many scholars showed special interest while thousand publications on the matter populated the web. That explains the reason why, among many other scholars, (Kloppmann, D., F., G., & D., 2004) in 2004, attested that BPEL has become the industrial-wide adopted service composition standard. This affirmation was followed by (Tim, Matthew and Bernd, 2008) in 2008 who confirmed that BPEL is indeed the de facto standard for business process modeling in today's enterprises and is a promising candidate for the integration of business and Grid application. In 2009 (Jonathan, Ying-Yan, Shang-Pin, & Shin-Jie, 2009) also attested that web service technologies are best exploited by composing services, and BPEL was adopted industry-wide as the de facto service composition standard. However, it is important to note that this first admiration for BPEL has since decreased. In fact, it can be observed that late publications on BPEL technology date back to 2010 and the number of publications have progressively reduced. It is also important to note that the last version of BPEL is outdated and dates back to 2007. Seven years after the publication of the second version (WS-BPEL 2.0) by (OASIS - BPEL Technical Committee, 2007), no other version for enhancements and improvement was designed. And in 2007, (Downey, 2007) stated that *"an execution language is not the right way to manage orchestration"* and showed that BPEL is so constrained in its capabilities that almost all vendors have had to extend it to make it do more than just simple orchestrations, and these extensions destroy any potential value that BPEL might provide as a "portable" process execution language (Downey, 2007). By pointing out that a BPEL process cannot be taken out of one engine and plugged into another one makes the language useless. That is to explain that, a standard which doesn't enable interoperability or portability is worse and useless. In a nutshell, the new language here defined by BPEL lost its first admiration. As a consequence, there are needs in exploring possible reasons and obstacle that led to a sinking of BPEL. Also, solutions to the obstacles in order to give back to BPEL its first admiration needed to be explored.

According to the BPEL specification (OASIS - BPEL Technical Committee, 2007), the language's goal is to provide "a notation for specifying business process behavior based on web services”. Defined entirely in XML, BPEL relies most fundamentally on a process element that contains various other elements. The language also has the ability to create compensation handlers for failed long-running transactions and a way to define correlation sets for associating responses with earlier requests. These fundamental advantages explain in one hand the reason why the language becomes all the more important when it was first published. On the other hand, if BPEL defines process using only XML data and communicates with other software only through web services, what about all of the other capabilities a real process needs to accomplish. The capabilities defined here express BPEL limitations. They include:

- BPEL inability to access local objects most likely written in Java or .NET language such as C#;
- BPEL’s inadequacy to communicate with other software outside its own environment. The communication is only achieved when the software is accessible via Web Service. Yet much software today doesn’t support Web Services;
- Also, BPEL’s incapability in defining a standard way to access relational data or other non-XML information. BPEL is focused entirely on Web Services access and allows working with XML data. However, processes commonly need to access data stored in relational database systems too.

These and many others limitations might be the reason for the BPEL’s decline. As a matter of fact, in-depth analysis needs to be carried out for it is important to understand what these limitations imply.

In addition to these challenges, (W.M.P, A.H.M, Kiepuszewski, & A.P., 2002) in 2002 and (W.M.P, Arthur and Nick, 2011) in 2011 carried out research whose aims were to analyse Workflow Patterns and they showed that BPEL specification does not provide full support for some Workflow Patterns. This might also be one of the possible reasons that explains BPEL’s decline. In short, despite BPEL’s first admiration, it seems clear that BPEL challenges are leading the language to an inevitable downfall.

In fact, BPEL challenges will not vanish in one go, this work focus on the latter problem identified by (W.M.P, Arthur and Nick 2011) which refers to BPEL's inability to support some Workflow Patterns. Workflow Pattern includes indeed, control flow pattern, data pattern, resource pattern, presentation and exception handling pattern. Based on this observance, our work has been motivated to finding out how support can be provided for BPEL to support unsupported Workflow Patterns. However, this project limits its scope to control flow patterns. In fact, control flow patterns are part of a business process in any organization; these patterns are classified from simple to complex which need all to be addressed by any business language. However, the current version of BPEL does not provide a direct support for capturing some of the control flow patterns. Though some work-around solutions have been proposed for some patterns, they have not been implemented in the current version of BPEL (BPEL version 2.0 in 2007). This suggests that further research, modeling and conceptualization have to be conducted to design a complete and robust business language.

1.3. Problem Statement

Business Process Execution Language (BPEL) was a powerful language which leverages software services to model and execute application as workflow processes. Any workflow language that is used to compose and manage business processes involving multiple Web Services must meet some key requirements (Onyeka & Sadjadi, 2005) such as:

i. The ability to adequately represent the business logic of the process;
ii. The ability to provide asynchronous as well as synchronous invocations of Web Services;
iii. The ability to support long-running transaction; and
iv. The ability to manage failures, exceptions and recovery.

The current version of WS-BPEL (WS-BPEL 2.0, 2007) specification attempts to meet these requirements, although some of the available features have certain limitations such as limited data manipulation. In addition, WS-BPEL 2.0 does not address some of the complex control flow patterns namely Multi-merge, Discriminator, Arbitrary Cycles, Milestone, Publish/Subscribe and Broadcast patterns which sometimes are important part of organizations' business. The organizations then find difficulties in representing all their business processes using the Business Process Execution Language (BPEL). Thus, there is a need to define activities, rules and attributes to support such patterns. However, it may be argued that it is not necessary to extend the language, since it is purely intended for orchestration and does not actually implement business logic components. However we propose in this work that the language can be improved upon in a way that allows business managers to compose business processes while permitting more experienced programmers to transform such processes into more robust and efficient solutions. All the aforementioned inability and inadequacy of BPEL to support complex control flow patterns effectively limit the language as a dynamic and flexible language for Web Service integration. In fact, architects who decide to use BPEL as their business language kick against BPEL ineffectiveness to support complex control flow within their organization. This often constrains the architects to abandon this level of the implementation or to abandon the automation of their business logic. This work therefore aims at extending the current BPEL version 2.0 from the root of its specification to support complex control flow and allow architects to effectively compose their business logic using BPEL.

1.4. Aim and Objectives of the Study

The aim of this work is to extend the BPEL language to support complex control flow patterns. The specific objectives are to:

i. Propose xBPEL as an extension of BPEL to support complex control flow patterns;
ii. Implement a prototype of xBPEL;
iii. Propose an xBPEL engine architecture for allowing xBPEL integration into existing BPEL engines.

1.5. Methodology

In order to attain the laid down objectives, results from existing works were used as the basis of our project. In fact, a thorough literature review was done in various concepts surrounding BPEL. Their characteristics and how they relate to BPEL was detailed. Thereafter, a study of related works in the area of Web Services was carried out. Work-around solutions for allowing BPEL to support complex control flow pattern were identified and an analysis of each solution is presented. For the implementation of the said extension, we focused on using Java Architecture for XML Binding (JAXB) and Java API for XML Web Service (JAX-WS). JAXB and JAX-WS are two libraries provided in Java SE for allowing the reading of XML files into their object representation and their writing vise-versa. This architecture refers to an Object-XML Architecture for XML files are converted into their object representation and their object back into XML files. In addition, a "start from WSDL" programming model was used. This approach focuses on existing XML Schemas document (WSDL) to generate necessary artifacts for communicating with the Web Service Providers. Using these artifacts, a prototype for each unsupported control flow pattern was implemented. Finally, based on logical gateway principle, we defined rules that need to be applied for each construct and proposed an architectural representation of a viable xBPEL engine.

1.6. Scope of the Study

The work encompasses three aspects:

The first aspect is the review of concepts surrounding BPEL itself. With it, we can have a clear understanding of how BPEL works. This review is based in W3C Recommendation and reports relevant to the technology.

The second aspect is the identification of common Workflow Patterns not supported by BPEL; or any other control flow patterns that BPEL does not provide a direct support to capture them. This second aspect is followed by a review of work-around solutions proposed by researchers as remedy to these shortcomings of BPEL and their evaluation.

Finally, the third aspect of this work is to develop an extension of BPEL, namely xBPEL in order to support these complex control flow patterns. This consists of developing a set of activities, rules and attributes to transform unsupported control flow patterns into BPEL processes. These set of activities will be defined in an abstract way, so that they can be reused and implemented in some other tools supporting Business Process Modeling and Notation (BPMN) and BPEL (ActiveBPEL, ActiveOS, Oracle SOA Suite …).

1.7. Structure of the work

Chapter 2 will provide background and details on BPEL specification in versions 1.1 and 2.0 and highlight control flow patterns. Chapter 2 will in addition give a brief overview of technologies surrounding BPEL, such as Web Services; Service Oriented Architecture, as well as the Web Service Description Language (WSDL) and the Universal Description, Discovery and Integration (UDDI). This chapter ends by surveying work-around solutions proposed by other researchers to support control flow patterns, their shortcoming and complexity and reviews related work to our proposed solution.

Chapter 3 will provide the methodology used to achieve the said objectives. The methodology is summarized in figure 3.1 and consists of identifying the control flow patterns not supported by BPEL 2.0, proposing the BPEL's extension and evaluating the proposed extension using formal models of evaluation.

Chapter 4 describes the proposed activities, constructs, rules, attributes and constraints necessary to support complex message patterns in BPEL. Once again, the set constructs, rules and attributes will be defined in an abstract way, so that they can be reused and implemented by BPEL engine vendors.

Finally, the Chapter 5 will summarize the achieved work, and future directions including ideas beyond the scope of this work.

CHAPTER TWO
REVIEW OF LITERATURE

2.0. Introduction

In this chapter, we give a structured overview of related works. Relevant works come from the field of Web Services, Service Oriented Architecture (SOA), Simple Object Access Protocol (SOAP), Web Service Description Language (WSDL) and Business Process Execution Language (BPEL). As illustrated in Figure 2.1, the key to achieving a good understanding of how BPEL works is its layered architecture. The dependency relation between each two adjacent layers is top-down. While the process layer (BPEL) takes care of the composition of Web services, the service layer (WSDL) provides a standard for describing the service interfaces. At the messaging layer (typically SOAP), the operations defined in the service layer are realized as two related output and input messages, which serialize the operation and its parameters. At the bottom of the stack is the transport layer, typically Hypertext Transfer Protocol (HTTP), Simple Mail Transfer Protocol (SMTP), File Transfer Protocol (FTP) and Block Extensible Exchange Protocol (BEEP) that facilitate the physical interaction between the Web services. Although Web services are independent of transport protocols (HTTP, SMTP, FTP, BEEP), HTTP is the most commonly used protocol for Web service interaction. Except for the transport layer, all the protocols in the other layers are typically based on the eXtensible Markup Language (XML).

In the rest of this chapter, we present a brief description of WSDL, SOAP and HTTP. This description serves the purpose of providing an overview about the mode of operation of each protocol in the other layers. In the context of better understanding of the underlying concept as related to our work, there is a need to discuss all concepts surrounding BPEL. Figure 2.1 shows the BPEL protocol stack and the surrounding concepts.

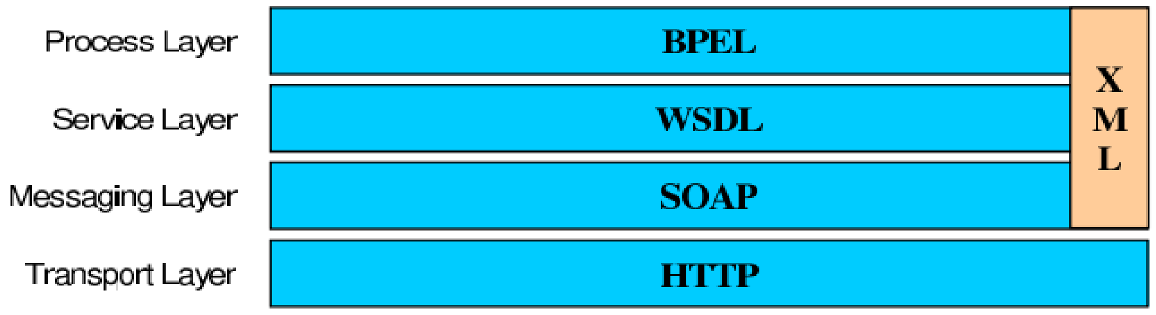


Figure 2.1: The BPEL protocol Stack (ONYEKA E. et al 2005)

2.1. Underlying Concepts

Before discussing the challenges relevant to message patterns in WS-BPEL, we briefly describe the basic technologies relevant to our approach, namely Service Oriented Architecture (SOA), the Business Process Execution Language for Web Services (WS-BPEL, BPEL4WS or BPEL for short) and control flow patterns.

2.1.1 Service Oriented Architecture (SOA)

The Service-Oriented Architecture (SOA) is an architectural paradigm which has gained great momentum in the industry in recent years. The Service-Oriented Architecture is the latest approach to building, integrating, and maintaining complex enterprise software systems. SOA is defined as:

"A paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations." – Organization for Advancement of Structured Information Standards (OASIS) definition (OASIS SOA – Reference Model (RM), 2012).

The SOA-RM specification bases its definition of the SOA around the concept of "*need and capabilities"*, where SOA provides a mechanism for matching needs of service consumers with capabilities provided by service providers. In computing terms, SOA is a standards-based method of systems development and integration. A SOA is generally regarded as having the following basic characteristics (Weerawarana, et al. 2005; Alonso, et al. 2004; Newcomer, et al. 2004):

- The basic building blocks of a SOA are *services*, which are loosely coupled and, mostly distributed;
- Services are described in some sort of *abstract interface language*, and can be invoked without knowledge of the underlying implementation;
- Services can be *dynamically discovered and used;*
- A SOA supports integration, or *composition*, of services.

In a nutshell, the main difference between SOA and BPEL is that SOA is an architectural concept and not a technology. BPEL is a technology to build SOA architectural programs. SOA concentrates on business rather that development or coding. Figure 2.2 shows the Web Service Architecture stack and the place of the BPEL technology in the architectural concept.

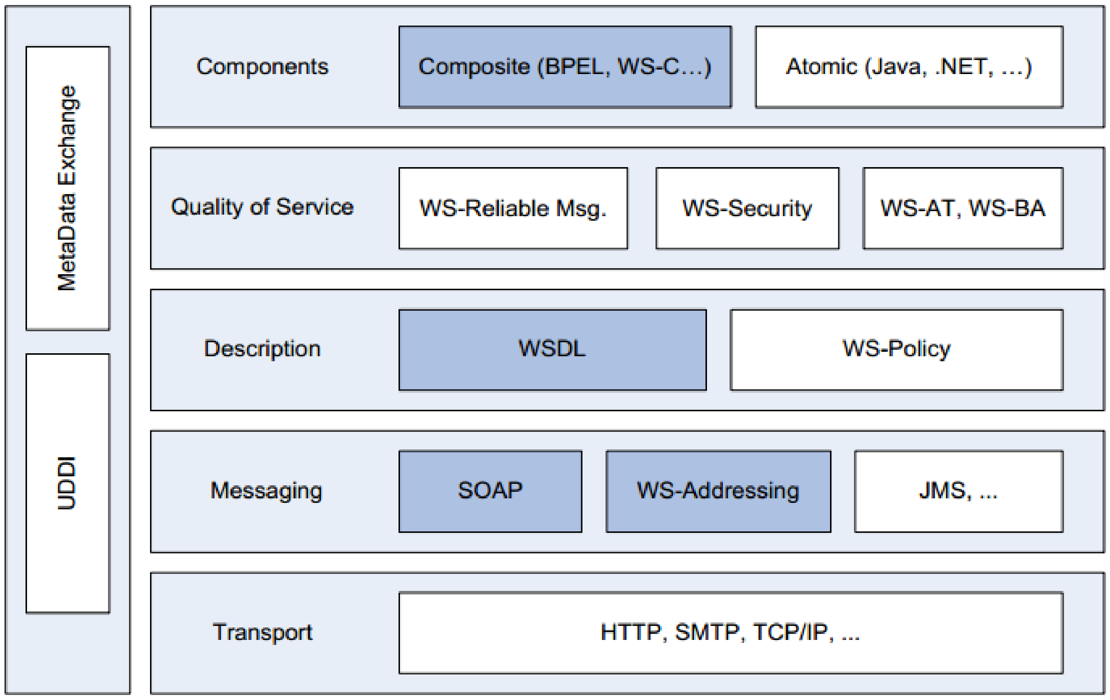


Figure 2.2: The Web Service Architecture Stack (Philip 2006)

This work is concerned with Web Service Composition, specifically with compositions written in the Business Process Execution Language (BPEL), which is located at the top of the diagram. BPEL depends on WSDL and SOAP (Tony, et al., 2003); additionally, the WS-Addressing specification is required to enable asynchronous messaging. Therefore, these three specifications will be explained in details in the next sections.

2.1.1.1 Web Services

Web Service is seen as an application *accessible* to other applications over the web. It is a method of communications between two electronic devices over the World Wide Web. It is a software function provided at a network address over the web with the service *always on* as in the concept of utility. There are many other definitions of a Web service. The Working Group W3C has jointly come to agreement on the following working definition which is relatively abstract:

"A software application identified by a URI, whose interfaces and bindings are capable of being defined, described, and discovered as XML artifacts. A web service supports direct interactions with other software agents using XML-based messages exchanged via Internet-based protocols". (W3C Working Group, 2004a).

The second definition includes the name of some of the most important W3C service standards, and stated as followed:

"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 (especially 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" (W3C Working Group, 2004b).

The latter definition is quite accurate and also hints at how Web Services should work. The definition stresses that Web service should be capable of being *"defined, described, and discovered*", thereby clarifying the meaning of *accessible* and making more concrete the notion of "*Internet-oriented, standards-based interfaces*". It also states that Web services should be "*services*" similar to those in conventional middleware. Not only should they be "*up and running*", but they should be described and advertised so that it is possible to write clients that bind and interact with them. In a nutshell, Web services are components that can be integrated into more complex distributed applications. This definition is very much in line with the perspective we take in this work compared with the first definition. A more precise definition is also provided by the UDDI consortium (UDDI Consortium, 2001), which characterizes Web service as "*self-contained, modular business applications that have open, Internet-oriented, standards-based interfaces*". This definition is also detailed, since emphasis is placed on the need for being compliant with Internet standards.

To summarize the definitions, a Web Service refers to any service that:

- Is available over the Internet or private (Intranet) networks
- Uses a standardized XML messaging system;
- Is not tied to any one operating system or programming language;
- Is self-describing via a common XML grammar;
- Is discoverable via simple find mechanism;

The Web service movement began with the publication of the three initial Web Service standards in mid to late 2000 (Philip 2006) which still forms the basic triangle of the Web service architecture (see Figure 2.3):

- **SOAP:** initially an acronym for Simple Object Access Protocol, is the basic messaging standard of Web services;
- **WSDL**: the Web Service Description Language, is an interface description language for Web services; and finally,
- **UDDI**: the Universal Discovery, Description, and Integration Service for Web services, handles dynamic discovery of Web Services.

Figure 2.3 depicts the Web Service architecture triangle (Publish – Find – Bind). This triangle briefly shows that Service Providers (SPs) describe the service they provide into a WSDL file and then ***publish*** this document into a registry (UDDI). A Service

Consumer can then use this registry to ***find*** a WSDL document and use this as an interface to ***bind*** itself with the Services.

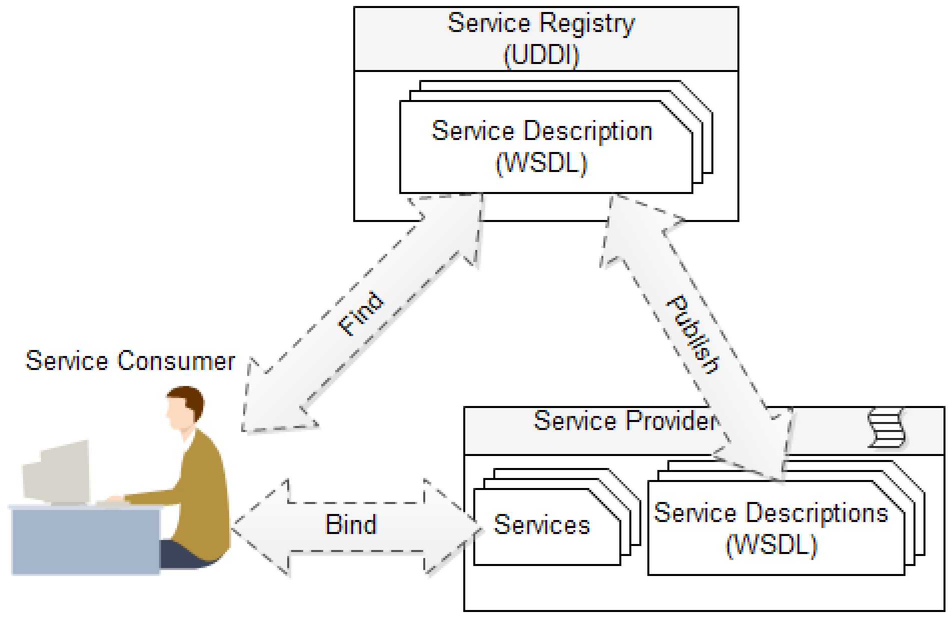


Figure 2.3: Web Services triangle

2.1.1.2 SOAP

SOAP (World Wide Web Consortium - SOAP, 2007) is a lightweight protocol intended for exchanging structured information in a decentralized environment. It uses XML technologies to define an extensible messaging framework providing a message construct that can be exchanged over a variety of underlying protocols. The framework has been designed to be independent of any particular programming model and other implementation specific semantics. SOAP specifies exactly how to encode an HTTP header and an XML file so that a program in one computer can call a program in another computer and pass information into it.

SOAP was initially created in 1999 (Philip, 2006) by Microsoft, DevelopMentor, and Userland, and stood for Simple Object Access Protocol. After being revised with contributions from IBM and Lotus, SOAP 1.1 was submitted to W3C for standardization, where it came to be known as SOAP (without being an acronym anymore) (World Wide Web Consortium - SOAP, 2007).

Figure 2.4 shows a basic structure of a SOAP message.

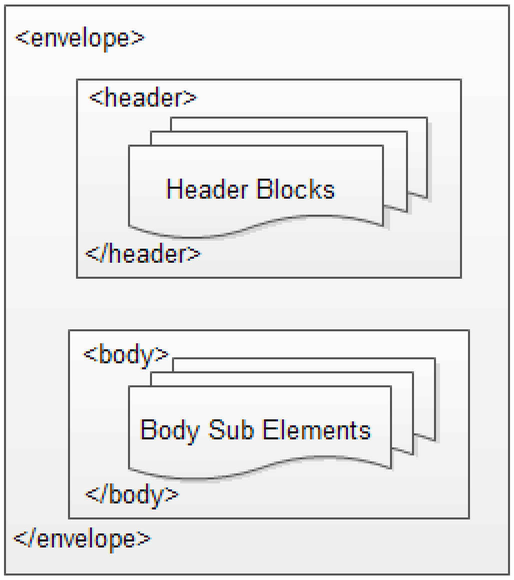


Figure 2.4: Structure of a SOAP Message

As described in the Figure 2.4, a SOAP message consists of a SOAP envelope, which encloses a SOAP header and a SOAP body. The SOAP header is optional, i.e. can be omitted. The SOAP body is mandatory, i.e. every valid SOAP message must contain a body. A SOAP body provides a mechanism for transmitting information to an ultimate SOAP receiver. It is an area where Web Services place the actual application data.

Listing 2.1 shows an example of SOAP message (World Wide Web Consortium - SOAP, 2007). It contains SOAP Header with a single SOAP header block and the mandatory SOAP Body.

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
<env:Header xmlns:env="http://www.w3.org/2003/05/soap-envelope" >
   <t:Transaction xmlns:t= "http://example.org/2001/06/tx"
                     env:mustUnderstand="true" > 5
   </t:Transaction>
</env:Header>
<env:Body>
   <m:alert xmlns:m="http://example.org/alert">
        <m:msg>Define the message here</m:msg>
   </m:alert>
 </env:Body>
</env:Envelope>
```

Listing 2.1: A SOAP Header & SOAP Body

2.1.1.3 Web Service Description Language (WSDL)

Definition 2.1: An **endpoint** is an association between a binding and a network address, specified by a Universal Resource Identifier (URI) that may be used to communicate with an instance of a service. An **endpoint** indicates a specific location for accessing a service using a specified protocol and data format (W3C Working Group, 2004b)

Definition 2.2: A **binding** is an association between an interface, a concrete protocol and a data format. A **binding** specifies the protocol and data format to be used in transmitting message defined by the associated interface (W3C Working Group, 2004b).

Definition 2.3: An **interface** also known as **service interface** is a logical grouping of operations. An **interface** represents an abstract service type, independent of transmission protocol and data format (W3C Working Group, 2004b).

Definition 2.4: An **operation** is a set of messages related to a single Web service action.

The Web Services Description Language (WSDL) is a W3C recommendation and an XML-based language to describe Web Services. W3C defines WSDL as an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information (Erik, Francisco, Greg, & Sanjiva, 2001). The ***operations*** and ***messages*** are described abstractly, and then ***bound*** to a concrete network protocol and message format to define an endpoint. WSDL is extensible to allow description of endpoints and their messages regardless of what message formats or network protocols are used to communicate (HTTP GET/POST, Multiplepurpose Internet Mail Extensions (MIME) or SOAP 1.1). WSDL is currently available in version 1.1, though version 2.0 is being drafted (Ambühler, 2005) and the latest version (2.0) of BPEL only supports WSDL 1.1. Details on WSDL 2.0 specification from the W3C can be found in (Roberto, Jean-Jacques, Arthur, & Sanjiva, 2007). For the purpose of this work, we shall only cover a few features of WSDL 1.1. Figure 2.5 describes a WSDL document and consists of two parts: An abstract part, and a concrete part. Both parts are enclosed in a global *definitions* block.

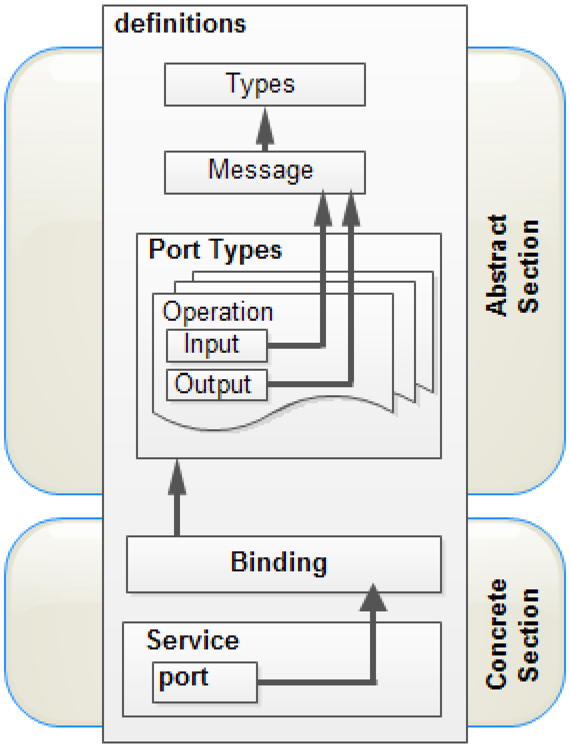


Figure 2.5: WSDL Document format

As pointed out in the above picture, WSDL document is simply a set of definitions. There is a definitions element at the root. This definition block contains a few attributes, among them the *targetNamespace (tns)* attribute. The value of this attribute can be viewed as the ID of this Web Service – although technically optional, the Web service cannot be properly used without it (Philip, 2006), as element within the definitions block need to be referenced by a qualified name.

The abstract part consists of three sections: the *types* element, an arbitrary number of *message* elements, and an arbitrary number of *portType* elements (Erik, Francisco, Greg, & Sanjiva, 2001). The following major element definitions are from W3C specification.

a) Types Element

The *types* element encloses data type definitions used to describe the messages exchanged. For maximum interoperability and platform neutrality, WSDL prefers the use of XML Schema Definition (XSD) as the canonical type system, and treats it as the intrinsic type system. Listing 2.2 shows an example of an inline XML schema definition inside a types block.

```
<types>
      <schema
       . . . .
        >
         <element name = "RequestInput">
                <complexType>
                  <sequence>
                        <element name = "input" type = "xsd:string">
                  </sequence>
                </complexType>
         </element>
</types>
```

Listing 2.2:WSDL Types element

b) Message Element

Message element is the basic data package which may be sent to the Web service, or sent by the Web service. They are specified by means of the *message* construct whereby each message has a name by which it is later referenced. Briefly, it is an abstract, typed definition of the data being communicated and consists of one or more logical parts as shown in Listing 2.3.

```
<message name = "RequestInputMessage">
      <part name = "body" element = "tns:RequestInput" />
</message>
<message name = "RequestPriceMessage">
      <part name = "product" type = "xsd:string" />
      <part name = "amount" type = "xsd:float" />
</message>
```

Listing 2.3: WSDL Message Section

c) Port type Element

A port type element is a named set of abstract operations and the abstract messages involved (Erik, Francisco, Greg, & Sanjiva, 2001). In other words it is an abstract set of operations supported by one or more endpoints.

```
<portType name = "InputMessagePT">
      <operation name = "GetInvoice" >
<input message = "tns:RequestInputMessage" />
        <output message = "tns:RequestPriceMessage" />
      </operation>
</portType>
```

Listing 2.4: WSDL Port Type Section

WSDL has four transmission primitives known as types of operation that an endpoint can support:

i. **One-way:** The endpoint just receives a message from a client of the service, not returning anything.
ii. **Request-response:** it is a typical Web service interaction whereby a message is received from a client and an answer is generated and sent back to the client.
iii. **Solicit-response:** here, the service itself initiates the operation by sending out a message and a message is returned.
iv. **Notification:** more similar to the one-way operation than to others, notification allows the endpoint to send the message without expecting an answer.

Briefly, a port type can be seen as an interface containing a list of operations which may be invoked on a service. In fact, it is difficult to imagine how the service might know about where to send its messages to without being contacted first. Asynchronous callbacks however require additional callback information to be sent along with the calls. The WS-Addressing standard described in section 2.1.1.4 offers the necessary infrastructure for such callbacks.

d) Binding Element

The binding element defines message format and protocol details for operations and messages defined by a particular *portType* (Erik, Francisco, Greg, & Sanjiva, 2001). It is a concrete protocol and data format specification for a particular port type. There may be any number of bindings for a given *portType* and the binding references the *portType* that it binds using the *type* attribute. The binding is defined using the SOAP binding extension as shown in Listing 2.5.

```
<binding name='NameOfTheBinding' type='tns:QNameOfThePortType'>
  <soap:binding style='rpc' transport='http://schemas.xmlsoap.org/soap/http'/>
  <operation name='GetInvoice'>
    <soap:operation soapAction=''/>
    <input>
      <soap:body use='encoded' namespace=''
        encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'/>
    </input>
    <output>
      <soap:body use='encoded' namespace=''
        encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'/>
    </output>
  </operation>
</binding>
```

Listing 2.5: WSDL Binding Section

e) Service Element

A service element is a collection of related endpoints (Erik, Francisco, Greg, & Sanjiva, 2001). It can been seen as the complete representation of a Web service, specifying which ports are available at which address, and how to invoke the operation inside the referenced *portType* by means of the given binding.

```
<service name='NameOfTheService'>
  <port name='NCNamePort' binding='tns:NameOfTheBinding'>
    <soap:address location='http://localhost:8080/nusoap/server1.php'/>
  </port>
</service>
```

Listing 2.6: WSDL Service Section

With the description of the *service* element, the concrete part of a WSDL document is complete, thus also completing the explanation of WSDL itself. However, the current version of WSDL is WSDL 2.0. Not only the meaning of the acronym has changed from version 1.1 where the **D** stood for **Definition**, but also it is now a fully W3C recommendation. WSDL 1.2 was renamed WSDL 2.0 because of its substantial differences from WSDL 1.1. Figure 2.6 highlights the differences and similarities.

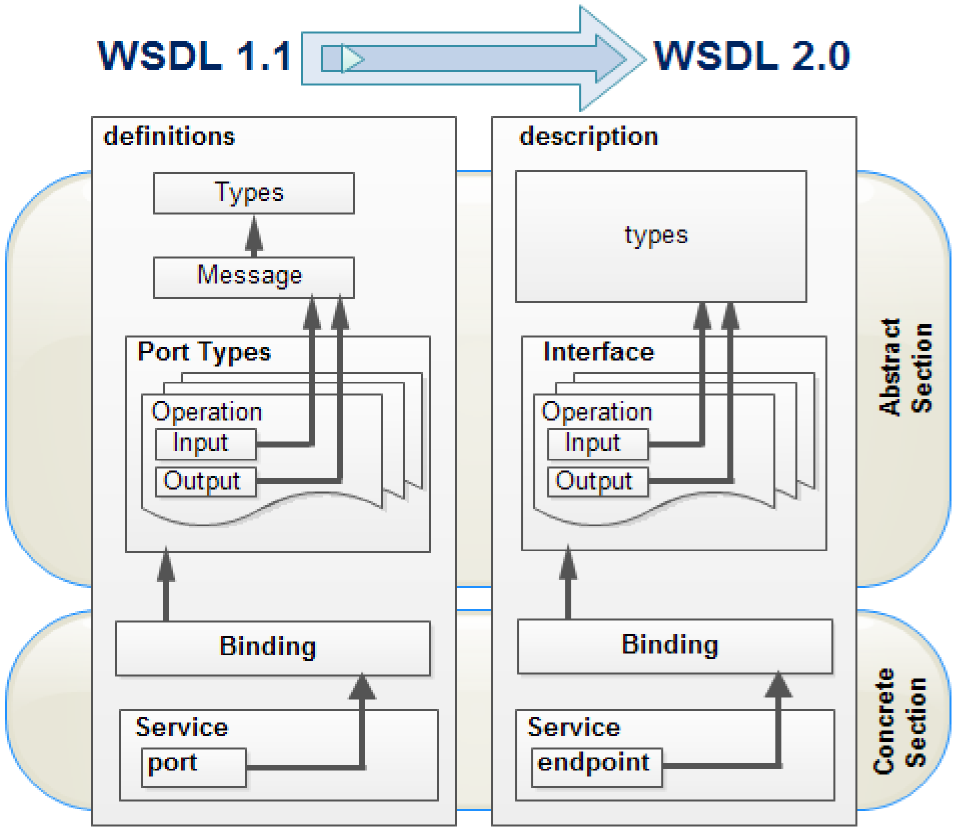


Figure 2.6: Comparison and analogies between WSDL 1.1 and 2.0 structures (adapted from Cristcost 2007)

Table 2.1 also presents in tabular form the differences between WSDL 1.1 and WSDL 2.0.

WSDL 1.1	**WSDL 2.0**	**Description**
Service	Service	Contains a set of system functions that have been exposed to the Web-based protocols
Port	Endpoint	Defines the address or connection point to a Web service. It is typically represented by a simple HTTP URL string
Binding	Binding	Specifies the interface and defines the SOAP binding style (RPC/Document) and transport (SOAP Protocol). It also defines the operations
PortType	Interface	Defines a Web service, the operations that can be performed, and the messages that are used to perform the

		operation.
Operation	Operation	Defines SOAP actions and the way message is encoded. Similar to a method or function in traditional programming language.
Messages	n/a	Contain the information needed to perform the operation. They were removed in WSDL 2.0 in which XML schema types for defining bodies of inputs, outputs and faults are referred to simply and directly
Types	Types	Describes the data. XML Schema Definition (XSD) is used for this purpose

Table 2.1: WSDL 1.1 and 2.0 terms and descriptions

2.1.1.4 WS-Addressing specification

As stated in the previous sections, routing messages to the correct Web service and, in the case of asynchronous callback, back to the original sender or different Web service, require additional addressing information to be included in the exchanged messages. The Web Services Addressing (WS-Addressing for short) is a standardized way of including message routing data within SOAP headers. It provides transport-neutral mechanisms to address Web services and messages (W3C Web Services Addressing - Core, 2006) and allows Web services to communicate addressing information.

The WS-Addressing specification was specially designed to deal with the following three characteristics of Web service communications (Weerawarana, Curbera, Leymann, Tony, & Ferguson, 2005):

- **Protocol independence:** Web services may use any kind of protocol, including proprietary protocols for communication. Furthermore, in its path to the ultimate receiver, a message may even be transferred by multiple protocols.

- **Asynchronous communication:** Web service message exchanges are not limited to synchronous request-response patterns, and may carry additional parameters which specify transactional properties.
- **Stateful, long-running operations:** The interactions between Web services are potentially stateful and long-running. The messages of such interactions are logically related and should be identified as such to be processed in the same environmental context. To decouple the lifetime of the SOAP request/response interaction from the lifetime of the HTTP request protocol, the service provider transmits the response message over a separate connection to the endpoint. This enables long-running interactions that can span arbitrary periods of time.

To provide a solution for all these requirements, WS-Addressing provides a protocol independent and an extensible mechanism for defining endpoints and including addressing information in messages. WS-Addressing may be used with any protocol; however, a special binding for SOAP is part of the specification (W3C Web Services Addressing - Core, 2006). Also, although WS-Addressing is part of the messaging layer and this one layer beneath service description (see Figure 2.2); it contains strong ties to WSDL as stated in (W3C Web Services Addressing - Metadata, 2007). This specification defines how the abstract properties defined in Core are described using WSDL, how to include WSDL metadata in endpoint references, and how WS-Policy can be used to indicate the support of WS-Addressing by a Web service. Details about how WS-Addressing relates to WSDL can be found in (W3C Web Services Addressing - Metadata, 2007). Listing 2.7 is taken from (W3C Web Services Addressing - Core, 2006) and describes the use of WS-Addressing in a SOAP 1.2 message.

```
(01) <S:Envelope xmlns:S="http://www.w3.org/2003/05/soap-envelope"
               xmlns:wsa="http://www.w3.org/2005/08/addressing">
(02)   <S:Header>
(03)    <wsa:MessageID>http://example.com/6B29FC40-CA47-1067-B31D-
00DD010662DA</wsa:MessageID>
(04)    <wsa:ReplyTo>
(05)      <wsa:Address>http://example.com/business/client1</wsa:Address>
(06)    </wsa:ReplyTo>
(07)    <wsa:To>http://example.com/fabrikam/Purchasing</wsa:To>
(08)    <wsa:Action>http://example.com/fabrikam/SubmitPO</wsa:Action>
(09)   </S:Header>
(10)   <S:Body>
(11)     ...
(12)   </S:Body>
(13) </S:Envelope>
```

Listing 2.7: use of message addressing properties in a SOAP 1.2 message

In Listing 2.7, the message is being sent from http://www.example.com/business/client1 to http://www.example.com/fabrikam /purchasing. Lines (02) to (09) represent the header of the SOAP message where the mechanisms defined in the specification are used. The body is represented by lines (10) to (12). Lines (03) to (08) contain the message addressing header blocks. Specifically, line (02) specifies the identifier for this message and lines (04) to (06) specify the endpoint to which replies to this message should be sent as an endpoint reference. Line (07) specifies the address URI of the ultimate receiver of this message. Line (08) specifies an action URI identifying expected semantics. This listing also completes the description of WS-Addressing as it relates to SOAP, thus to BPEL.

2.1.1.5 Universal Description, Discovery and Integration (UDDI)

The OASIS UDDI specification defines UDDI by its functions. It stated in (OASIS UDDI Specification, 2004) that "*the focus of UDDI is the definition of a set of service supporting the description and discovery of (1) businesses, organizations, and other Web services providers, (2) the Web services they make available, and (3) the technical interfaces which may be used to access those services*". Based on a common set of industry standards, including HTTP, XML, XML Schema, and SOAP, UDDI provides an interoperable, foundational infrastructure for a Web services-based software environment for both publicly available services and service only exposed internally within an organization. In our own understanding, UDDI is an XML-based registry and a

Web-based distributed directory that enables businesses on the Internet to discover each other. Its ultimate goal is to streamline online transactions by enabling companies to find one another on the Web and make their systems interoperable. Founded by Microsoft, IBM and Ariba, UDDI is sponsored by the Organization for the Advancement of Structured information Standards (OASIS).

2.1.2. Business Process Execution Language for Web Services (WS-BPEL)

WS-BPEL is an XML-based language and an orchestration language used to describe execution logic of Web services application by defining their control flows and providing a way for partner services to share a common context (Yuli, 2007). In fact, BPEL leverages other Web service standards such as SOAP and WSDL for communication and interface description and describes the inbound and outbound process interfaces in WSDL so that they can be easily integrated into other processes or applications. In turn, this allows consumers of a process to inspect and invoke a BPEL process just like any other Web service. Despite the relatively complex definition, BPEL is a very useful tool for explicit description of company processes that can be used for further automatic execution.

The WS-BPEL process itself is a kind of flow-chart, where each element in the process is called an *activity*. An activity is either a primitive or a structured activity. The set of the *primitive activities* contains: *invoke,* invoking an operation on some Web service; *receive,* waiting for a message from external source; *reply,* replying to an external source, *wait,* waiting for some time; *assign,* copying data from one place to another; *throw,* indicating errors handlers in the execution; *terminate,* terminating the entire service instance, and *empty,* doing nothing. And the set of *structured activities* includes *while*, *if….elseif, RepeatUntil.* BPEL was designed around concepts which have been described above (SOAP, WSDL and UDDI) and addresses the following goals:

- Define business processes that interact with external entities using web services and at the same time have a web interface of their own. The process itself thus becomes a service. This is how processes can be composed into larger assemblies.

- Define business processes using an XML-based language. BPEL does not define a graphical representation of processes or provide any particular design methodology for processes; it just standardizes their XML definition. BPMN does the graphical representation.
- Provide data manipulation functions for simple manipulation of data needed to define process data and control flow.
- Support the implicit creation and termination of business process instances as the basic lifecycle mechanism. The process can also be suspended, resumed, etc.
- Define a long-running transaction model that supports transaction scoping and failure recovery when the transaction is not finished successfully.
- Use web services as the model for process decomposition and assembly.
- Build on web services standards.
- Support event-driven process management (asynchronous call model)

Briefly, the language is intended to support the modelling of two types of processes: executable and abstract processes (Petia, Wil M.P., & Marlon, 2002). An *abstract*, (not executable) *process* is a business protocol, specifying the message exchange behaviour between different parties without revealing the internal behaviour for anyone of them. While an *executable process* specifies the execution order between a numbers of *activities* constituting the process, the *partners* involved in the process, the *messages* exchanged between these partners, and the *fault* and *exception handling* specifying the behaviour in cases of error and exceptions. The next sections will give technical as well as structural overview of BPEL.

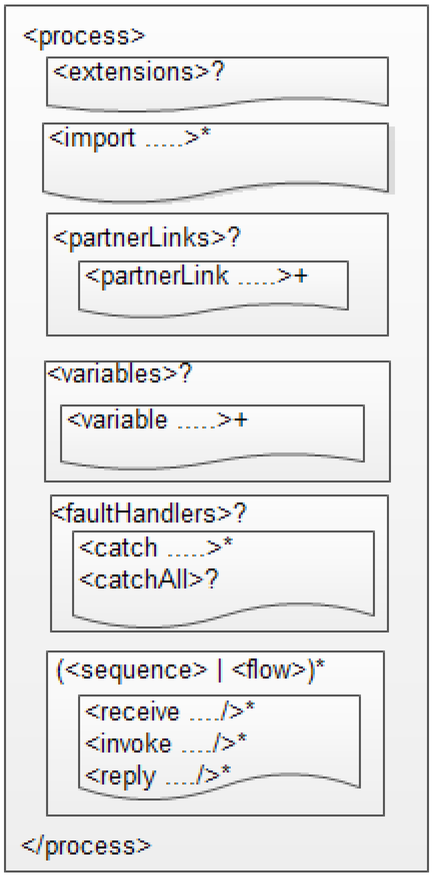


Figure 2.7: Basic Structure of a WS-BPEL document

2.1.2.1.Structure of a WS-BPEL Definition

Graphically, the general structure of a WS-BPEL process definition might look like the following figure (Figure 2.7). It represents an XML document containing WS-BPEL language constructs performing the process logic.

A WS-BPEL as well as some other documents are deployed to a WS-BPEL engine against which that process definition will be executed. The following table lists the most important WS-BPEL constructs used when defining WS-BPEL definitions (Yuli, 2007):

Constructs	Description
Process	Represents the root element of a WS-BPEL process and uses attributes to declare a number of the process-related namespaces
partnerLinks	Contains set of partnerLink elements. Each partnerLink element **establishes the relationship between the process service** and one of its partners.
Variables	Contains set of variable elements. Each variable element defines a variable used by the process
faultHandlers	Contains fault handlers that describe the actions to be taken in response to the faults that may arise during the process execution
Sequence	Contains one or more activities performed one after another, as they appear within the construct. In a complex scenario, several sequences may be grouped within the flow construct, which enables concurrency and synchronization.
flow	Enables concurrency between the activities enclosed within this construct.
Receive	Wait for a message to be received from a partner, it specifies partner from which message is to be received, as well as the port and operation provided by the process used by the partner to pass the message
Invoke	Issue an asynchronous request. Synchronously invoke an in-out operation of a web service provided by a partner

Reply	Can be synchronous or asynchronous. It is a response to a request corresponding to a receive activity. A combination of Receive/Reply corresponds to request-response operation in WSDL
Assign	Contains the from and to pairs wrapped in a copy element, which are used to update the values of variables defined within the variables section
If	Allows adding conditional logic to the process definition. Note: *if/elseif/else* is new WS-BPEL 2.0. In BPEL4WS 1.1, we use *switch/case/otherwise* instead.
Throw	Throws a named fault inside the business process.

Table 2.2: basics WS-BPEL constructs

Listing 2.8 shows a complete structure of the BPEL language

```
<process name="NCName" targetNamespace="anyURI"
   queryLanguage="anyURI"?
   expressionLanguage="anyURI"?
   suppressJoinFailure="yes|no"?
   exitOnStandardFault="yes|no"?
   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable">
   <extensions>?
      <extension namespace="anyURI" mustUnderstand="yes|no" />+
   </extensions>
   <import namespace="anyURI"?
      location="anyURI"?
      importType="anyURI" />*
   <partnerLinks>?
      <!-- Note: At least one role must be specified. -->
      <partnerLink name="NCName"
         partnerLinkType="QName"
         myRole="NCName"?
         partnerRole="NCName"?
         initializePartnerRole="yes|no"?>+
      </partnerLink>
   </partnerLinks>
   <messageExchanges>?
      <messageExchange name="NCName" />+
   </messageExchanges>
   <variables>?
      <variable name="BPELVariableName"
         messageType="QName"?
         type="QName"?
         element="QName"?>+
         from-spec?
      </variable>
   </variables>
```

```
<correlationSets>?
      <correlationSet name="NCName" properties="QName-list" />+
   </correlationSets>
   <faultHandlers>?
      <!-- Note: There must be at least one faultHandler -->
      <catch faultName="QName"?
         faultVariable="BPELVariableName"?
         ( faultMessageType="QName" | faultElement="QName" )? >*
         activity
      </catch>
      <catchAll>?
         activity
      </catchAll>
   </faultHandlers>
<eventHandlers>?
      <!-- Note: There must be at least one onEvent or onAlarm. -->
      <onEvent partnerLink="NCName"
         portType="QName"?
         operation="NCName"
         ( messageType="QName" | element="QName" )?
         variable="BPELVariableName"?
         messageExchange="NCName"?>*
         <correlations>?
            <correlation set="NCName" initiate="yes|join|no"? />+
         </correlations>
         <fromParts>?
            <fromPart part="NCName" toVariable="BPELVariableName" />+
         </fromParts>
         <scope ...>...</scope>
      </onEvent>
      <onAlarm>*
         <!-- Note: There must be at least one expression. -->
         (
         <for expressionLanguage="anyURI"?>duration-expr</for>
         |
         <until expressionLanguage="anyURI"?>deadline-expr</until>
         )?
         <repeatEvery expressionLanguage="anyURI"?>
            duration-expr
         </repeatEvery>?
         <scope ...>...</scope>
      </onAlarm>
   </eventHandlers>
   activity
</process>
```

Listing 2.8: A complete structure of BPEL language

2.1.2.2.History of BPEL

BPEL emerged as a combination of two prior, competing XML languages for composition: the IBM Web Service Flow Language (WSFL) and Microsoft XLANG (Weerawarana, Curbera, Leymann, Tony, & Ferguson, 2005). Although similar in their goals, each language has a different composition approach.

In WSFL, the flow model is a business process described as a directed graph of activities (the nodes of the graph) and control connectors (the edges of the graph). Nodes and edges of the graph are annotated with attributes, such as transition conditions, that determine the execution of the model. WSFL also provides the possibility to define a global model, in which the overall partner interactions can be specified. A global model is a simple recursive composition meta-model that describes the interactions between existing Web services and defines new Web services as a composition of existing ones. While XLANG provides a notation "for the specification of message exchange behavior among participating Web services" so that "automation of business processes based on Web services" can be achieved. To achieve this objective, the major constructs of XLANG include sequential and parallel control flow definitions, long-running transactions with compensation, custom correlation of messages, exception handling, and dynamic service referral. An XLANG service description extends a WSDL service description with the behavioral aspects of the service. It allows the user to specify a set of ports in a service section of a WSDL document and extend it with a description of, for example, the sequence in which the operations that the ports provide are to be used.

In BPEL, processes are created by using a combination of the graph-oriented style of WSFL and the algebraic style of XLANG. BPEL also allows to recursively compose Web services into new aggregated Web services. The first version of the BPEL4WS specification was released in 2002. Version 1.1, released in 2003, was submitted to OASIS for standardization. The group is set to produce the next version, renamed WS-BPEL version 2.0, as its first output. However, BPEL4WS v2 (drafted since 2004) does not support WSDL 2.0.

2.1.2.3.BPEL Partner Links and WSDL Port Types

BPEL represents all communication links – whether incoming or outgoing – as abstract *partner links.* From the process's point of view, a partner link is simply a communication channel which allows the initiator to send exactly one request, and the recipient to reply with at most one response (Gregory, Christopher, & Gruia-Catalin, 2007). Each partner link is bound to a single remote endpoint at a time. The current version of BPEL does not allow processes to inspect or modify a partner link's binding, except by copying bindings directly between two partner links. The following picture describes the relationship existing between a WS-BPEL and WSDL documents.

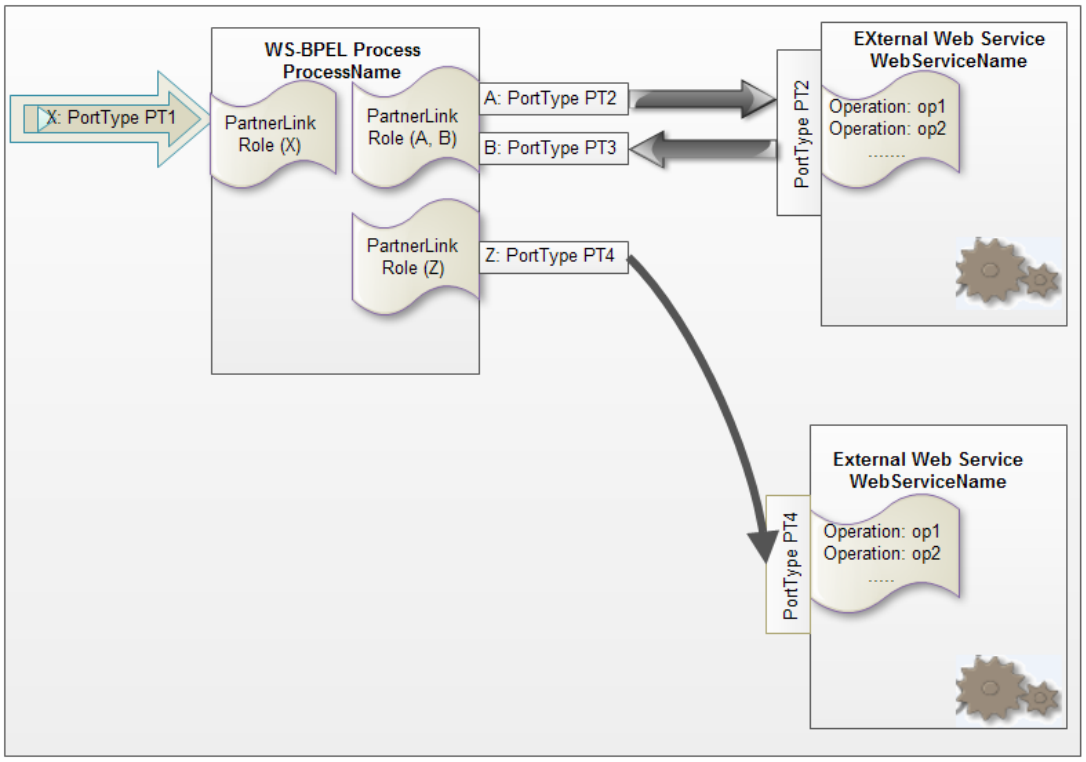


Figure 2.8: Relationship between WS-BEPL Partner Links and Port Types (Allen, 2008).

The following conclusions are derived from the above relationship: (i) the process offer services and uses services from other partners and WSDL port types specify the interfaces between partners. (ii) Partner link types are a WSDL extension, and defined within the scope of the WSDL definitions of the process. (iii) Each partner link type defines one or two roles, which are performed by the partners. Each role is typed by a WSDL port type.

2.1.2.4.BPEL 2.0

WS-BPEL 2.0 is the current version of BPEL under development by OASIS (OASIS - BPEL Technical Committee, 2007). The current version of BPEL 2.0 has some changes from version 1.1 that are worth mentioning here:

- New activity types such as *repeatUntil*, validate, *forEach* (parallel and sequential), *rethrow*, *extensionActivity*, *compensateScope*.
- Renamed activities: switch/case renamed to if/else, terminate renamed to exit.
- Termination Handler added to scope activities to provide explicit behavior for termination.
- Variable initialization.
- XML schema variables in Web service activities.
- Locally declared *messageExchange* (internal correlation of receive and reply activities).
- Clarification of Abstract Processes (syntax and semantics).
- Enable expression language overrides at each activity.

For this project, BPEL 2.0 is the main focus. For more details about BPEL 2.0, OASIS specification can be consulted at (OASIS - BPEL Technical Committee, 2007).

2.1.2.5.Business Process Model and Notation (BPMN 2.0)

The WS-BPEL standard defines the meaning and features of particular activities, their XML definition; however, it does not specify their graphical representation. Therefore, some providers have created their own graphic notation of BPEL that is being improved constantly, so that the processes are as clear as possible. Thus, BPMN (Business Process Model and Notation) is a graphic notation used to model processes in a clear and illustrative way.

The goal of BPMN is to provide companies with the possibility to map their processes and see them in graphical representation. The appearance and meaning of all the symbols are standardized so that everyone can understand what a given model does and how the process works. This means a major improvement when there is a need to discuss the processes across the company or between business and departments.

The difference between BPEL and BPMN is clear:

- BPMN is a descriptive notation for modeling business processes that uses standardized graphical symbols, meanings and logical connections.

- BPEL is a standard for executing and describing processes and it uses more technical tongue. It is an XML-based language.

So in a nutshell, company uses BPMN to describe and model the processes and then convert the given BPMN construction into BPEL, however, there is not always a simple mapping between BPMN and BPEL and that is why attempts to generate BPEL processes from the BPMN notation often have more or less failed although not impossible. BPMN is supported by a lot of commercial and open source implementations with varying levels of support for turning diagrams into code, among them Bonita BPM Community, Eclipse JWT (Java Workflow Tooling); Oracle BPEL PM and Bizagi Process Modeler etc…

2.1.3. Control Flow Patterns

This section describes common control flow patterns used in organizations. Web services composition and workflow management are related in the sense that both are concerned with executable processes. Therefore, much of the functionality in workflow management systems is also relevant for the Web Services composition languages like BPEL, XLANG, and WSFL (Petia, Wil M.P., & Marlon, 2002). In this section, we consider the 20 workflow patterns presented in (W.M.P, A.H.M, Kiepuszewski, & A.P., 2002) and revised in (W.M.P, Arthur, & Nick, Workflow Patterns Initiative, 2011). A summary of a pattern based analysis of BPEL4WS are also presented in Table 2.3. It shows how and to what extent these patterns can be captured in WS-BPEL. A comprehensive set of patterns (data, control, resource, presentation and exception handling patterns) can be found in (W.M.P., Arthur, & Nick, 2011) which is a pattern initiative that describes the different ways of dealing with patterns in the context of some process-aware information systems.

A '+' in a cell of the table (Table 2.3) refers to direct support (i.e. there is a construct in the language which directly support the pattern). A '-' in the table refers to no direct support. Sometimes there is a feature that only partially supports a pattern, e.g., a construct that implies certain restrictions on the structure of the process. In such cases, the support is rated as '+/-'.

Control Flow Patterns		**Product/Standard**		
		BPEL	**XLANG**	**WSFL**
1	Sequence	+	+	+
2	Parallel Split	+	+	+
3	Synchronization	+	+	+
4	Exclusive Choice	+	+	+
5	Simple Merge	+	+	+
6	Multi Choice	+	-	+
7	Synchronizing Merge	+	-	+
8	**Multi Merge**	**-**	**-**	**-**
9	**Discriminator**	**-**	**-**	**-**
10	**Arbitrary Cycles**	**-**	**-**	**-**
11	Implicit Termination	+	-	+
12	MI Without Synchronization	+	+	+
13	MI with a Priori Design Time Knowledge	+	+	+
14	**MI with a Priori Runtime Knowledge**	**-**	**-**	**-**
15	**MI without a Priori Runtime Knowledge**	**-**	**-**	**-**
16	Deferred Choice	+	+	-
17	**Interleaved parallel Routing**	**+/-**	**-**	**-**
18	**Milestone**	**-**	**-**	**-**
19	Cancel Activity	+	+	+
20	Cancel Case	+	+	+
Communication patterns				
21	Request/Reply	+	+	+
22	One-way	+	+	+
23	Synchronous Polling	+	+	+
24	Message Passing	+	+	+
25	**Publish/Subscribe**	-	-	-
26	**Broadcast**	-	-	-

Table 2.3: Comparison of BPEL, XLANG, and WSFL Workflow using both workflow and communication patterns (W.M.P, Arthur and Nick 2011)

Details on message patterns can be found in technical reports (Petia, Wil M.P. and Marlon 2002; W.M.P, A.H.M, et al. 2002; W.M.P, Arthur and Nick 2011). However, for the purpose of this work, we shall only cover patterns not supported by BPEL.

The following conclusions can be made from the table:

i. The first five patterns correspond to the basic routing constructs; they are naturally supported by all languages.
ii. WS-BPEL as a language integrating the futures of the block structured language XLANG and the directed graphs of WSFL indeed supports all patterns supported by XLANG and WSFL since they correspond to subsets of the WS-BPEL (see section 2.1.2.2).
iii. WS-BPEL as Web Service Composition language provides constructs for communication modelling which clearly distinguishes it from traditional workflow modelling languages.

Besides these positive remarks, there are also three negative comments concerning WS-BPEL. First of all, WS-BPEL is a complex language because if offers (too) many overlapping constructs. The simple fact that many of the patterns can be realized using “XLANG style” and “WSFL style” illustrates its complexity. Secondly, the semantics of WS-BPEL is not always clear. The precise semantics of advanced concepts like serializable scopes leave room for multiple interpretations thus, complicating the adoption of the language. Finally, no direct supports are provided for some of the patterns. The latter concerns the problem which this work intends to address.

In order to grasp the activities, rules and attributes proposed in this work to solve the above problem (see section 1.3) a description of these control flow patterns not supported is indispensable. The next sections present a description of these patterns.

2.1.3.1.Workflow Patterns

This section describes workflow patterns not supported by WS-BPEL according to Table 2.3.

a) Multi-merge (WP1)

Description: The multi merge pattern is used to model the convergence of two or more branches into a single path. It is a point in the workflow process where, based on a decision or workflow control data, a number of branches are chosen (W.M.P, A.H.M, Kiepuszewski, & A.P., 2002). Practically, it is the convergence of two or more branches into a single subsequent branch such that each enablement of an incoming branch results in the thread of control being passed to the subsequent branch. Basically, what comes after the multi-merge will be executed one for each incoming replies.

Example: the "*lay foundation"*, "*order material*" and *"book laborer*" tasks occur in parallel as separate process branches. After each of them completes the *"quality review*" task is run before that branch of the process finishes. Another concrete example may be: during a recruitment process, it is necessary to check the references of the future employee. It is very important to check the personal and professional references provided by the employee. Each time a reference is checked, the Human Resources Manager must be notified.

b) Discriminator (WP2)

Description: This pattern describes a point in the process that waits for one of the incoming branches to complete before activating the subsequent activity. The other incoming branches are omitted after being completed. Once all the incoming branches are completed the discriminator is reset (W.M.P, A.H.M, Kiepuszewski, & A.P., 2002). As soon as one of the two concurrent branches replies, the flow resumes and the other branch 'forgotten' (continues its execution without the parent process caring for its outcome).

Example: to improve query response time, a complex search is sent to two different databases over the internet. The first one that comes up with the result should proceed the flow. The second result is ignored.

c) Arbitrary cycles (WP3)

This pattern models a point in a workflow process where one or more activities can be done repeatedly (W.M.P, A.H.M, Kiepuszewski, & A.P., 2002). It is a point where a portion of the process (including one or more activities and connectors) needs to be 'visited' repeatedly without imposing restrictions on the number, location, and nesting of these points.

Example: The deliverables of a project need approvals from several people. The number of approval needed is a number defined by the project manager.

d) Interleaved parallel routing (WP4)

Description: WP4 occurs when a set of activities is executed in an arbitrary order (Petia, Wil M.P., & Marlon, 2002). Each activity in the set is executed exactly once. The order is decided at runtime: it is not until one activity is completed that the decision on what to do next is taken and no two activities are executed at the same moment (i.e. no two activities are active for the same workflow instance at the same time).

Example: at the end of each year, a bank executes two activities for each account: *"add interest"* and "*charge credit card costs"*. These activities can be executed in any order. However, since they both update the account, they cannot be executed at the same time.

e) Milestone (WP5)

Description: WP5 is the situation whereby the enabling of an activity depends on the case being in a specified state, i.e. the activity is only enabled if a certain milestone has been reached which did not expire yet (W.M.P, A.H.M, Kiepuszewski, & A.P., 2002). A milestone is, indeed defined as a point in the process where a given activity A has finished and an activity B following it has not yet started.

Example: After having placed a purchase order, a customer can withdraw it at any time before the shipping takes place. To withdraw an order, the customer must complete a withdrawal request form, and this request must be approved by a customer service representative. The execution of the activity "*approve order withdrawal"* must therefore follow activity "*request withdrawal"*, and can only be done if: (i) the activity "*place order"* is completed, and (ii) the activity "*ship order"* has not yet started.

The two remaining workflow patterns namely MI with a Priori Runtime Knowledge (WP6) and MI without a Priori Runtime Knowledge (WP7) are not discussed in this work.

The description of control flow patterns not supported by WS-BPEL therefore put an end to section 2.1. In the next section, related works are discussed and critical look is cast on work-around solution provided by those previous researches.

2.2. Related Works

This section presents works that are related to our project. This project is built on concept in SOA (Service-Oriented Architecture), Web service, business orchestration and control flow patterns.

2.2.1. Extending BPEL

Researchers have recognized that augmenting BPEL's capabilities is the key to building more flexible and interoperable web services (Gregory, Christopher and Gruia-Catalin 2007; Tim, Matthew and Bernd 2008; Michiel, et al. 2009). This section discusses relevant work concerning the extension of BPEL and its adaptability to other environment.

First, in this category is the work done by Gregory, et al. 2007 in "*Extending BPEL for Interoperable Pervasive Computing*". Gregory, et al. pointed out that BPEL's inflexible communication model effectively prohibits its deployment on the kind of dynamic wireless networks used by most pervasive computing devices. They presented extensions of BPEL that address these restrictions, transforming BPEL into versatile platform for interoperable pervasive computing applications. BPEL assumption that the number of participants (partner links) in the application is known at design time and cannot be changed at runtime does not fit typical pervasive computing settings, where participants may come and go at any time. Despite this limitation BPEL processing constructs are highly expressive. To address the issue the proposed extensions focus on communication pattern and make the following changes:

1) Processes may declare *partner groups*, or partner links that are bound to multiple incoming endpoints simultaneously.
2) Processes may send multicast message to all members in a partner group.
3) Processes may send or receive an arbitrary number of messages over a partner link or partner group.
4) Processes may make limited changes to the bindings of partner links, with well-defined behavior.

Briefly four new BPEL activities or construct are presented in this paper and showed in Listing 2.9.

```
<ext:add partnerGroup = "ncname" partnerLink = "ncname" mustNotBeMember =
"no|yes"?  />
<ext:remove partnerGroup = "ncname" partnerLink = "ncname" mustNotBeMember =
"no|yes"?  />
<ext:close partnerLink = "ncname" />
<ext:unbind partnerLink = "ncname" />
<ext:reply (partnerGroup = "ncname" | partnerLink = "ncname") moreMessages =
"no|yes"? … />
```

Listing 2.9: Extensions proposed in (Gregory, Christopher and Gruia-Catalin 2007)

The BPEL extensions described in this paper which address multicast/broadcast is only limited to the area of pervasive computing also known as ubiquitous computing. Beside these extensions presented by Gregory, et al. 2007; Michiel, et al. 2009 in "*VxBPEL: Supporting variability for Web services in BPEL*" proposed an approach, VxBPEL to deal with variability in service-centric (SC) systems. They provided a prototype implementation to interpret VxBPEL. They introduced several activities (keywords) that allow variability information, especially variation points, variants and realization relations to be modelled. Allen 2008, in "*Providing Context in WS-BPEL Processes*" presented an extension of WS-BPEL based on the notion of a context variable. He described how the WS-BPEL language-extension mechanism can be used to design context variables, and how these variables can be used in business processes. Another extension in BPEL was also provided by Tim, et al. 2008, in "*Composition and Execution of Secure Workflows in WSRF-Grids*". In their paper, Tim, et al. 2008 extended BPEL with related security feature. In fact, the current BPEL security feature is not equipped to deal with complex multiprotocol Grid environments and does not integrate with the Grid Security Infrastructure (GSI). In this work therefore, a solution that extends the BPEL security approach to encompass secure Grid application interactions was presented. The additional extension to BPEL is introduced and defined as follows:

```
<gridInvoke … >
<security
      Method = "GSITransport |
                 GSISecureMessage |
                 GSISecureConversation"
     Level = "privacy | integrity"
     Authz = "none | selft | host | anyString"?
     Pre-credentials = "filename"?
     Anymous = "true | false"?
     Delegation = "none | full | limited"?
/>?
</gridInvoke>
```

Listing 2.10: Syntax of the security settings for invocation

Finally, a library which adds information to the message header of the client's SOAP call was introduced. The header of these two functions are showed in Listing 2.11

```
setProxyCertificate(Call call, String pathToProxyCert)
setCredentials (Call call, String myProxyHost, String username,
                               String passwd, in lifetime, Boolean
autoRenewal)
```

Listing 2.11: Functions' prototype for extending SOAP Header

Though BPEL language has been extended by security-related settings in this extension, the WS-BPEL specification doesn't provide direct support for security settings. It rather recommends the use of WS-Security to implement security features. In another paper titled "*Grid Workflow Modelling Using Grid-Specific BPEL Extensions*", Tim, et al. 2007 elucidated the problem concerning the invocation of WSRF-based services in Grid computing using BPEL4WS and also presented a solution to this problem by extending the BPEL specification. An implementation based on the ActiveBPEL workflow enactment engine was described.

Michael 2009, in his report "*Extending BPEL with transitions that can loop*" described an extension to WS-BPEL 2.0 to support transitions, which are similar to BPEL's links, except that unlike links, the directed graph can include cycles. With this extension, BPEL support the free-form control flow that is allowed by the BPMN 2.0 notation, without taking on all the complexity of BPMN 2.0 execution language. The aim of this report was to deal with BPEL's inability to allow cycles. The proposed extension slightly modifies the semantics for links and the resulting construct introduced is called "*transitions*".

Karastoyanova, et al. 2005 in "*Extending BPEL for run time adaptability*" addressed the problem of dynamic binding of Web Services (WSs) to WS-flow instances at run time i.e. the ability to exchange a WS instance participating in a WS-flow instance with an alternative one. This is becomes additionally complicated when we consider that the execution of process depends on its deployment and that once a process is deployed, WS instances cannot be exchanged. They, therefore described a "find and bind" mechanism by specifying criteria for port selection during process deployment, and showed its representation as a BPEL extension. A prototypical implementation of the presented functionality is also discussed. However, since deployment is execution environment specific it cannot be standardized in a cross-platform manner. Moreover, selection policies can only be modified when processes are redeployed. These limitations suggest that supports deployment-independent ports selection at run time on process instance level is needed. Besides, Decker, et al. 2007 in "*BPEL4Chor: Extending BPEL for Modeling Choreographies*" showed how BPEL can be extended for defining choreographies. The proposed extensions (BPEL4Chor) distinguish between three aspects: the participant behavior descriptions, the participant topology using message links and the participant groundings. Although only few constructs were added on the top of BPEL, BPEL4Chor does support Service Interaction Patterns but provides no support for Atomic Multicast Notification. Another prominent paper to our project is "*BPEL Extensions to User-Interactive Service Delivery"* where Jonathan, et al. 2009 proposed an extension to BPEL to infuse user interactions into composite services along three dimensions: (1) to develop two BPEL extension activities to describe the inner workings of user interactions in BPEL service and the rendering of service user interfaces; (2) to provide a wizard-style mechanism to guide the user to interact with the service flow in accordance with the sequence of service execution; and (3) to devise a UI service communication protocol to facilitate secure cross-domain communication among UI services from various domains. Also, Holmes, et al. 2008 presented the architecture of Vienna BPEL for People (VieBOP), a new BPEL4People system that can be coupled with an arbitrary BPEL engine.

To summarize the related works concerning extension on BPEL, we have identified papers, though not closely related to our extension, provided extension on concept surrounding WS-BPEL. As a matter of fact, König, et al. 2008 presented a liberal approach to decide compatibility between an abstract and an executable BPEL process; Deutch, et al. 2009 analysed the complexity of query evaluation over business processes and presented novel algorithms for computing top-k query result, that is finding the k most likely matches. Besides, Karastoyanova, et al. 2009 argued that applying the Aspect Oriented Programming (AOP) to WS and BPEL environments would boost process flexibility. They therefore presented the syntax for such aspect as an extension of the WS-Policy framework.

2.2.2. Work-around solutions

As pointed out in section 2.1.3, BPEL 2.0 provides no support for interleaved parallel routing pattern. However, a work-around solution using deferred choice (i.e. pick) as proposed in (W.M.P, A.H.M, Kiepuszewski, & A.P., 2002) can be applied (see Listing 2.12).

```
<pick>
  <onMessage m1>
    <sequence>
         Activity A
         Activity B
    <sequence>
  </onMessage>
  <onMessage m2>
    <sequence>
         Activity B
         Activity A
    </sequence>
  </onMessage>
</pick>
```

Listing 2.12: Interleaved parallel routing workaround solution (W.M.P, A.H.M, Kiepuszewski, & A.P., 2002)

The drawback of this solution is its complexity, which increases exponentially with the number of activities to be interleaved (Petia, Wil M.P., & Marlon, 2002).

It also is possible to capture this pattern in BPEL using the concept of serializable scopes (Petia, Wil M.P., & Marlon, 2002). A scope provides concurrency control in governing access to shared variables (OASIS - BPEL Technical Committee, 2007). A serializable scope is an activity of type scope whose *ContainerAccessSerializable* attribute is set to “yes”. The activities to be interleaved are placed in different containers which all write to a single shared container. Since the activities are in different containers, they can potentially be executed in parallel (see Listing 2.13).

```
<flow>
   <scope name = ‘s1’
      containerAccessSerializable := ‘yes’>
    <sequence>
         Write to container C
         Activity A
         Write to container C
    </sequence>
  </scope>
  <scope name = ‘s2’
      containerAccessSerializable := ‘yes’>
    <sequence>
         Write to container C
         Activity B
         Write to container C
    </sequence>
  </scope>
</flow>
```

Listing 2.13: Interleaved parallel routing work-around solution (Petia, Wil M.P., & Marlon, 2002)

Since the serializable scopes that contain the activities write to the same container, no two of them will be “active” simultaneously, but instead, they will be executed one after the other (Petia, Wil M.P., & Marlon, 2002). Three things are worth pointing out with respect to this solution. First, the semantics of serializable scopes in BPEL is not clearly defined (1). The BPEL specification only states that this semantics is “similar to the standard isolation level serialization of database transaction”, but it does not specifies where does the similarity stop (e.g. how does the underlying transaction model deal with or prevent serialization conflicts?). Second, it is not possible in this solution to externally influence (at runtime) the order in which the activities are executed: this order is instead fixed by the transaction manager of the underlying BPEL engine. Finally, since serialization scopes are not allowed to be nested, this solution is not applicable if one occurrence of the interleaved parallel routing pattern is embedded within another occurrence

Another and very simply, but unsatisfactory solution is to fix the order of execution i.e. instead of using parallel routing, sequential routing is used. Since the activities can be executed in an arbitrary order, a solution using a predefined order may be acceptable. However, by fixing the order, flexibility is reduced and the resources cannot be utilized to their full potential.

W.M.P, et al. 2002 also proposed a work-around solution for milestone pattern. A deferred choice between executing the activity B, or executing activity E, is made. A while loop is used to guarantee that as long as B is not chosen, E can be executed an arbitrary number of times. The limitation of this solution is that activity E cannot be restricted by any parallel threads (Petia, Wil M.P., & Marlon, 2002).

```
Activity A
B_completed := “false”
<while B_completed = “false”>
  <pick>
   <onMessage me>
         Activity E
   </onMessage>
   <onMessage mb>
         <sequence>
                 B_complete := “true”
         </sequence>
  </pick>
</while>
Activity B
```

Listing 2.14: Milestone work-around solution (W.M.P, A.H.M, Kiepuszewski, & A.P., 2002)

2.2.3. Analyzing Control patterns

This section describes works related to Workflow patterns. They are related to our work in the way that, they provide proof that WS-BPEL does not support all patterns. These papers simply analyzed commonly patterns as they relate to products and standards used in business process. These products/standards are, among others BPEL, XLANG, WSFL, Staffware and MQSeries.,Petia, et al. 2002 in “*Pattern Based Analysis of BPEL4WS*” established that little or no effort has been dedicated to systematically evaluating the capabilities and limitations of business languages and techniques. Their paper is presented as the first step towards this direction. It presents an in-depth analysis of the Business Process Execution Language for Web Service (BPEL4WS). The framework used for this analysis is based on a collection of workflow patterns. A summary of the results from the analysis is presented in Table 2.2. In their paper, they also showed a comparison of BPEL4WS with XLANG, WSFL and two major Workflow Modelling Languages: Staffware PLC's Staffware and IBM's MQSeries Workflow. On the whole, 20 workflow patterns and 6 communication patterns were analyzed.

They concluded that 7 among the 26 patterns are not supported by BPEL. Though no solution was proposed, this paper contributed a lot in identifying the unsupported pattern in BPEL4WS. Similar to this paper, W.M.P., et al. 2002 in "*Workflow Patterns*" presented the most detailed papers describing Workflow patterns. An overview of Workflow patterns was presented, emphasizing the control perspective, and discussing to what extent current commercially available workflow management systems could realize such patterns. Dusan 2013, addresses the issue of initial steps in the business process life cycle, mainly the simulation. He described the three selected BPMN 2.0-compliant business process management tools and compares their modeling and simulation feature. Finally, an innovative engine was proposed, designed and implemented by employing the characteristics of BPSim, a young simulation standard. Beside all these related works, a whole project, known as WS-Notification provides ways and means to support subscribe/publish pattern in a BPEL process.

CHAPTER THREE
METHODOLOGY

3.0. Introduction

This chapter describes the methodology used for identifying the control flow patterns and design pattern followed for extending BPEL. For the sake of consistency the extensions provided in this work are described with the tag xbpel. Therefore, all tags with xbepl are part of the extensions while other tags with the same notation used as in OASIS Standard (2007) are parts of the BPEL specification. The methodology of this work is outlined in figure 3.1

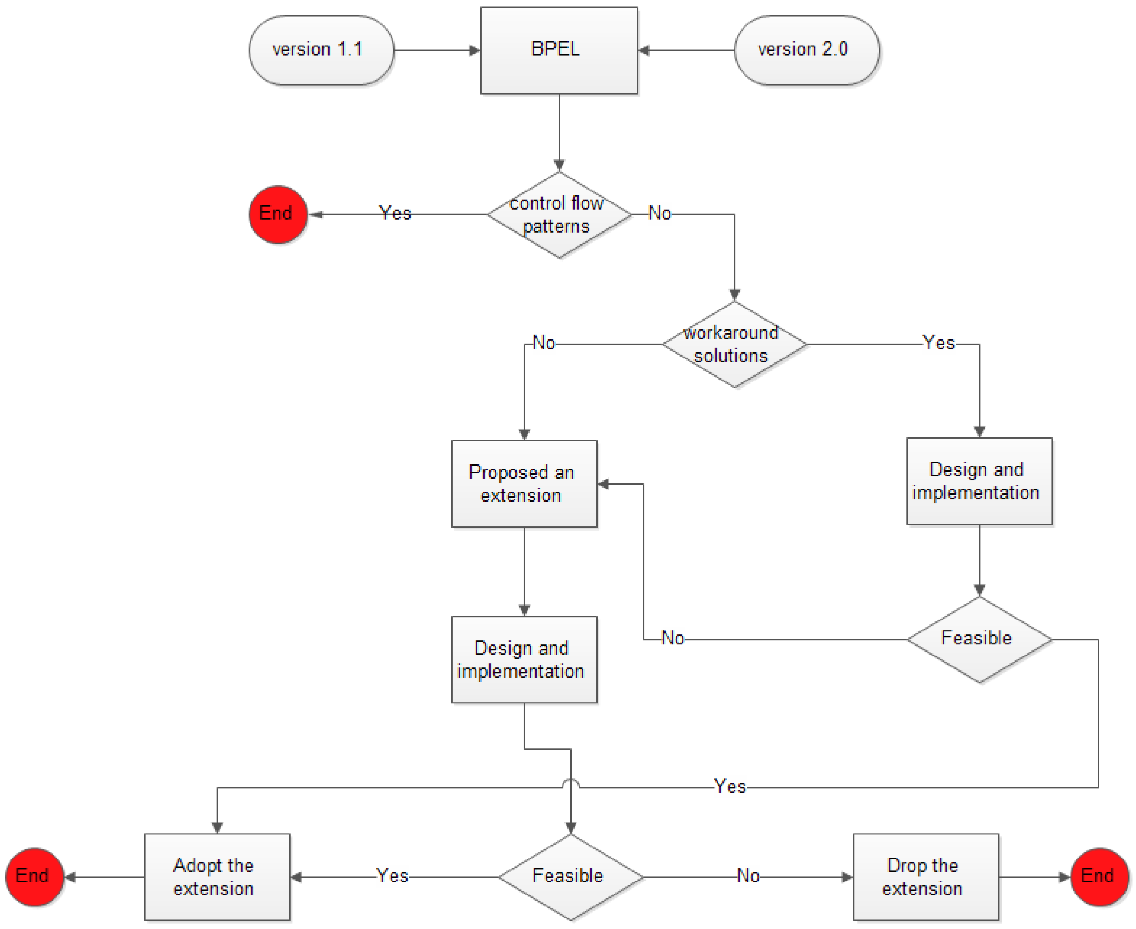


Figure 3.1: Skeleton of the methodology

To attain the laid down objectives, Java language was used to implement the business logic in the client side. In order to write/read XML documents from/to Java classes the Java API for XML Web Services (JAX-WS) and the Java Architecture for XML Binding (JAXB) were used. In the next section, details about the standards used to implement the prototype are presented.

3.1. Implementation and architectural environment

Java Enterprise Edition (Java EE) is a large framework comprising a number of specifications (Allen, 2008). It includes solutions for a number of applications such as persistence, transactions, web-application development, web-service development, etc. the prototype herein defined only uses a subset of the web services technologies provided by the framework. This section aims not at detailing all the inner workings of Java EE's web services used in the prototype, so only a broad outline is provided. Briefly, to implement the prototype, the following requirements are necessary:

- Execution Environment: Java SE 1.7 and Java EE 5.
- Java Runtime Environment (JRE): jre7
- Java Development Kit (JDK): jdk1.7 update 40
- Window 7 Home Premium; Copyright © 2009, Service Pack 1
- System: ASUSTek Computer Inc.
- Processor: Intel$^{(R)}$ Core™ i3 CPU M 370 @ 2.40GHz 2.40GHz
- Intalled memory (RAM): 4.00GB (3.79 GB usable)
- System type: 64-bit Operating System

In fact, in order to develop a web service application we need a Java EE platform instead of Java SE platform, however, since version 1.6 of Java SE, JAXB is part of the Java SE platform and still remains one of the APIs in the Java EE platform.

Definition 3.1: JSR stands for Java Specification Request. Changes to the Java platform can be proposed by members of the Java Community through the creation of a JSR.

In addition, an understanding of the following specifications is also necessary:

- JSR 224: The Java API for XML-Based Web Services (JAX-WS)
- JSR 222: The Java Architecture for XML Binding (JAXB)
- JSR 181: Web Services Metadata for the Java Platform (WS-Metadata)

Roughly speaking, the architecture for implementing the prototype is an Object-XML Architecture illustrated in Figure 3.2. The starting point is an XML schema document herein the BPEL extension xBPEL proposed in this work.

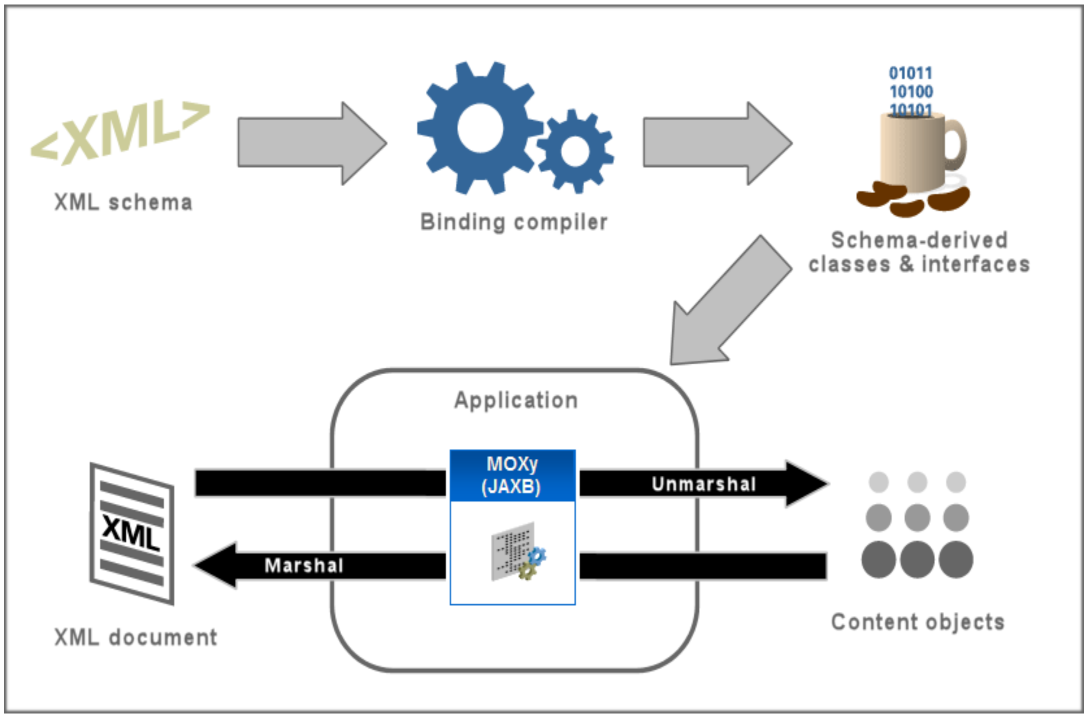


Figure 3.2: Object-XML Architecture (Eclipse - The Object-XML component, 2014)

The next sections describe these specification and show to what extends they are necessary for achieving the objective of this project.

3.2. JSR 224: Java API for XML Web Services (JAX-WS)

With the emergence of XML as the standard for exchanging data across disparate systems, Web Service applications need a way to access data that are in XML format directly from the Java application. Data binding describes the conversion of data between its XML and Java representations. There exist several types of data binding which are:

- JAXB: stands for Java Architecture for XML Binding. It binds class data and performs unmarshalling (reading) object from XML and marshalling (writing) by saving object in XML.

- Aegis: unlike JAXB, Aegis is not a standard specification, but plays a same role to JAXB as a function provided only in Apache CXF.
- MTOM: stands for Message Transmission Optimization Mechanism. It is an optimization mechanism of SOAP message transmission.

The reason why we prefer using JAXB for our data binding is because it is more widely used and accepted as the standard for data binding and is standard specification. Moreover, data in JAXB are transmitted in XML document while MTOM transmits data in attachment type that also requires additional setting to use MTOM.

JAX-WS uses Java Architecture for XML Binding (JAXB) to manage all of the data binding tasks. Specially, JAXB binds Java method signatures and WSDL messages and operations and allows customizing the mapping while automatically handling the runtime conversion. This makes it easy to incorporate XML data and processing functions in applications based on Java technology without having to know much about XML

3.3. JSR 222: Java Architecture for XML Binding (JAXB)

Java Architecture for XML Binding (JAXB) provides a fast and convenient way to bind XML Schemas and Java representations, making it easy for Java developers to incorporate XML data and processing functions in Java applications. In fact, JAXB provides methods for unmarshalling (reading) XML instance documents into Java content trees, and then marshalling (writing) Java content trees back into XML instance documents. It also provides a way to generate XML schema from Java objects. In pursuance of recall, the extension provided in this project is XML-based and conforms to the BPEL specification. That makes JAXB all the more significant for the project as it would allow the extension to be unmarshalled into Java content and marshal the Java content back into XML document, that is the xBPEL. The reason why JAXB is used to read and write the extension which is an XML-based syntax is because there is a need of testing the prototype. Since no BPEL engine actually understands the extension, for some of the constructs, attributes, rules thereafter defined are completely new to the. Therefore, by using JAXB, we ensure and prove that our extension is feasible and can be adopted by

BPEL engine vendor. In fact, most of the BPEL engines are written in Java and support WS-BPEL 2.0 specification. Since our prototype is also implemented in Java, it also adds credence to our proposed extension to support complex control patterns.

In a nutshell, the methodology such as depicted in Figure 3.1 consists of developing constructs, attributes, and rule using XML tags and JAXB.

3.3.1. JAXB Architecture

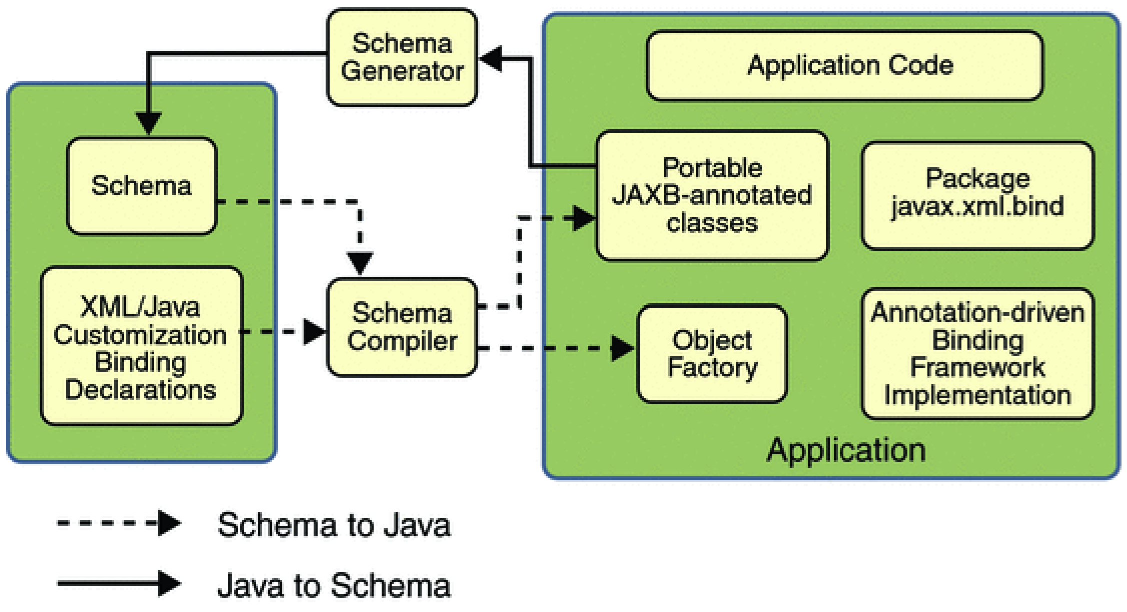


Figure 3.3: JAXB Architectural Overview (Oracle - JAXB Architecture, 2010)

Figure 3.3 shows the components that make up a JAXB implementation. In this figure, a JAXB implementation consists of the following components:

- **Schema compiler:** Binds a source schema to a set of schema-derived program elements. The binding is described by an XML-based binding language.
- **Schema generator:** Maps a set of existing program elements to a derived schema. The mapping is described by program annotations (See Section 3.3.2).
- **Binding runtime framework:** Provides unmarshalling (reading) and marshalling (writing) operations for accessing, manipulating, and validating XML content using either schema-derived or existing program elements.

3.3.2. JAXB data binding process

A data binding process in JAXB basically consists of two main functions: marshal and unmarshal.

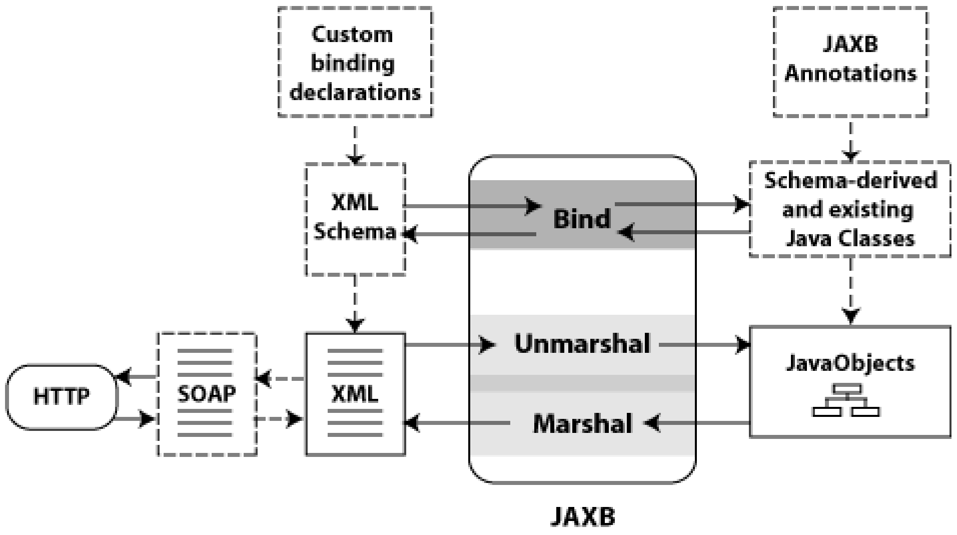


Figure 3.4: Data Binding with JAXB (Oracle, 2013)

Figure 3.4 shows the JAXB data binding process. In this figure, the JAXB data binding process consists of the following tasks:

- **Bind:** Binds XML Schema to schema-derived JAXB Java classes, or value classes. Each class provides access to the content via a set of java bean style access methods (get or set methods). Binding is managed by the JAXB Schema compiler (Oracle, 2013).
- **Unmarshal:** converts the XML document to create a tree of Java program elements, or objects that represents the content and organization of the document that can be accessed by Java code. In the content tree, complex types are mapped to value classes. Attribute declarations or elements with simple types are mapped to properties or fields within the value class and these values can be accessed using get and set methods. Unmarshalling is managed by the JAXB binding framework (Oracle, 2013)
- **Marshal:** It converts the Java Objects back to XML content. In this case, the Java methods that are deployed as WSDL operations determine the schema

components in the *wsdl:types* section. Marshalling is managed by the JAXB binding framework.

3.3.2. Java-to-XML Schema and XML-to-Java Mapping

The mapping between WSDL operations and Java methods, and XML Schema types and Java objects/types, is governed by the JAX-WS WSDL↔ Java mapping and JAXB XML↔ Java binding rules, respectively (Allen, 2008). These rules are detailed in JSR-224, JSR-222 and JSR-181. In addition, a full list of supported XML Schema data types and their corresponding Java data types can be found in (Oracle, 2013). Especially, for our prototype implementation, only built-in data types such as *integer*, *string*, *double* and *boolean* were used. Table 3.1 shows the mapping XML Schema Built-in Data Types to Java Data Types used in this project.

XML Schema Data Type	**Java Data Type**
xsd:string	java.lang.String
xsd:integer	java.math.BigInteger
xsd:Boolean	Boolean
xsd:double	Double

Table 3.1: Mapping XML Schema Built-in Data Types to Java Data Types

Where xsd refers to the target namespace http://www.w3.org/2001/XMLSchema, now and in the rest of the project

Moreover, the default binding rules for Java-to-XML Schema mapping can be overridden using JAXB annotations. Table 3.2 summarizes the JAXB mapping annotations that can be included in a Java Web Service file to control how the Java objects are mapped to XML. Each of these annotations is available in the *javax.xml.bind.annotation* package of the Java SE 1.7.

Annotation	Description
@XMLAccessorType	Specifies whether fields or properties are mapped by default.
@XMLElement	Maps a property contained in a class to a local element in the XML Schema complex type to which the containing class is

	mapped.
@XMLMimeType	Associates the MIME type that controls the XML representation of the property with a textual representation, such image/jpeg.
@XMLRootElement	Maps a top-level class to a global element in the XML Schema that is used by the WSDL of the Web Service.
@XMLType	Maps a class or enum type to an XML Schema type

Table 3.2: JAXB Mapping Annotations

Based on the Java API for XML (JAX) and the Java Architecture for XML Binding (JAXB), a consequent programming model to implement the prototype was designed. The next section gives details on the programming model or approach used in the prototype implementation and describes the basic steps for creating business logic or simply a web service in the client side.

3.4. Programming model

There are three major approaches to web service development (Allen, 2008):

- Start from WSDL: using this programming model, the XML Schemas exist and JAXB unmarshals the XML document to generate the Java objects. That is, the developer has a pre-defined WSDL and needs to write Java code to implement its operations
- Start from Java: Here, Java classes are created and at run-time, JAXB marshals the Java objects to generate the XML content which is then packaged in a SOAP message and sent as a Web Service request or response. In simpler words, the developer here starts with the implementation objects and annotates them as necessary. Thus, using the *public static Endpoint publisher (String, Object)* method from the abstract class Endpoint, the WSDL interface is generated automatically.
- Meet in the middle: in this approach, the developer has existing Java code, but needs to conform to a predefined WSDL. This is, by far, the hardest approach and involves a lot of iterative experimenting, tweaking annotations and code generation.

Therefore, the simplest way to create business client logic is by using the "start from WSDL" approach. The prototype defined in this project uses the "Start from WSDL" approach. In fact, WSDL documents available in publicly available Web Service Providers were first retrieved and saved in the client system for the implementation. Necessary XML-Schema types for each control flow patterns were also specified using JAX-WS annotations while necessary Java classes were generated using *wsimport* tool.

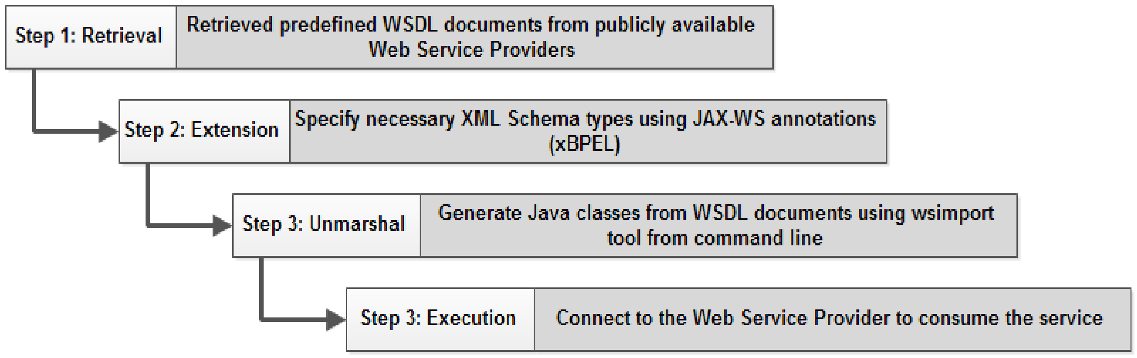

Step 1: Retrieval
Retrieved predefined WSDL documents from publicly available Web Service Providers
Step 2: Extension
Specify necessary XML Schema types using JAX-WS annotations (xBPEL)
Step 3: Unmarshal
Generate Java classes from WSDL documents using wsimport tool from command line
Step 3: Execution
Connect to the Web Service Provider to consume the service

Figure 3.5: Programming model used for the prototype implementation

Figure 3.5 shows the programming model used for the implementation of the prototype.

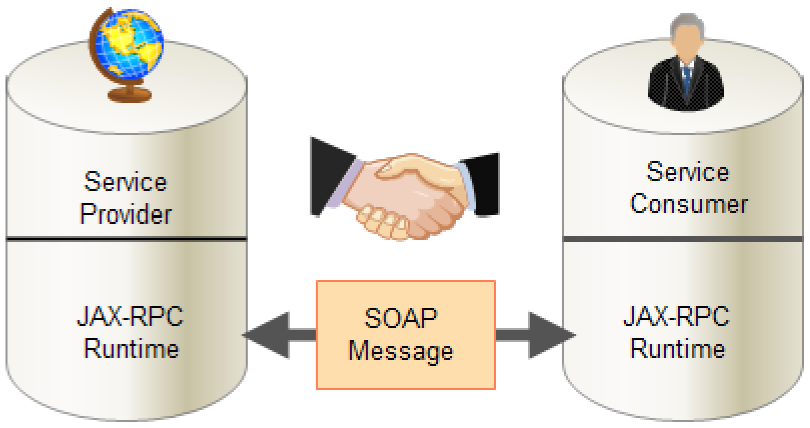


Figure 3.6: Communication between JAX-RPC Web Service and a Client

Figure 3.6 illustrates how Java API for XML – Remote Procedure Call (JAX-RPC) technology manages communication between a web service and client. The starting point for developing a JAX-RPC web service is the Service Endpoint Interface (SEI). In fact, A Service Endpoint Interface (SEI) is a Java interface that declares the methods that a client can invoke on the service. These methods are also described in a WSDL document. The SEI can either be coded from scratch or generated using *wsimport* (Since Java SE 1.6)

As stated above, in the introduction of this chapter, steps for creating a web service and client will be provided. The basic steps include:

1. Code the SEI and implementation class and interface configuration file.
2. Compile the SEI and the implementation class.
3. Use *wscompile* to generate the files required to deploy the service.
4. Use *deploytool* to package the files into a WAR file.
5. Deploy the WAR file. The tie classes (which are used to communicate with clients) are generated by the Application Server during deployment.
6. Code the client class and WSDL configuration file.
7. Use *wscompile* to generate and compile the stub files.
8. Compile the client class.
9. Run the client.

The implementation of a web service must therefore follow these basic steps. However, in this project, we tried to follow this approach in the implementation of the prototype but could not be made to work especially in step 3 and 7. This was due to the absence of the *wscompile* command in the Java Environment albeit the system variable JAVA_HOME was set to C:\Program Files\Java\jdk1.7.0_40. As a result of this, the implementation of the prototype skipped these two steps and has directly used the *wsimport* command to generate the necessary stub files, making it possible to attain the fixed objectives (see section 1.4).

In addition to JAXB and JAX specification, some knowledge of JSR 67: SOAP with Attachment API for Java (SAAJ) is also required. It is not crucial to have in-depth knowledge of the all the above JSRs – in fact, it is unlikely that any developer will require all the capabilities they provides. At minimum, however, the following concepts need to be known:

- ✓ The development process for implementing web services
- ✓ What artifacts (code, WSDL interfaces, XML schemas, etc.) are required

- ✓ What SOAP messages are mapped to Java methods
- ✓ The request/response framework
- ✓ The basics of how XML ↔ Java data binding works

Before rounding up this chapter, one of the key elements used to achieve the implementation is by using the concept of logical operators or gateways.

3.5. Logical gateways

Michael in (Hovey Michael 2005) showed that control flow patterns such as multi-choice, synchronization merge, multi-merge and discriminator are expressed as logical gateways and sometimes interchangeably used. For example, the multi-choice which describes the forking of a process to multiple branches is sometimes known as OR-split. This pattern differs from the Exclusive Choice pattern which is also known as exclusive-OR or XOR-split. Also, the synchronization pattern which joins branches spawned by a multi-choice is known as inclusive OR-join. These and more others examples that show that patterns can be achieved through logical operator led this project to use the same methodology to describe how rules and constructs for supporting complex control flow patterns can be implemented. In this section therefore, we give a brief description of various logical gateways and split constructs in the context of given a clear understanding to the next chapter which gives description of the proposed extension and its prototype implementation.

Here are the definitions of some common logical gateways:

Definition 3.2: AND-split diverges the thread of control in a given branch into multiple concurrent execution threads in several branches. That is after the execution of a given branch, all the subsequent branches must always occur.

Definition 3.3: XOR-split or eXclusive OR-split routes the thread of control in a given branch into one of several possible outgoing branches on the basis of an execution decision made at runtime. In simply terms, XOR-split executes only one of the subsequent branches after the execution of the given branch.

Definition 3.4: OR-split routes the thread of control in a given branch into one, several or all outgoing branches on the basis of an execution decision(s) made at runtime. It means that after the execution of the given branch, one or all of the subsequent tasks can be executed.

Definition 3.5: Thread-split diverges the thread of control in a given branch into multiple concurrent execution threads in the same branch. This construct causes the subsequent task to be executed several times.

Definition 3.6: AND-join represents a point in the business process under enactment by a workflow management system in which two or more activities which have simultaneously executed converge into a single control point. Each incoming branch executed in parallel is stopped until the set of all transitions to the next activity is completed.

Definition 3.7: OR-join is a point in the workflow modeling a business process where two or more activity branches re-converge to a single common activity as the next step within the workflow. As no parallel activity execution has occurred at the join point, no synchronization is required.

Thus, this section ends the description of the methodology used to achieve the laid down objectives which main aim is to provide support for complex control flow patterns in Business Process Execution Language (BPEL). The next chapter, which is the design and implementation of xBPEL concentrates on defining rules using logical gateways, proposing constructs using XML tags and defining attributes. These descriptions serve the basic for any BPEL engine integration.

3.6. Multithreading Programming

Multithreading programming is one of the key concepts used to achieve the proposed extension. Therefore, it is important to give a brief description of multithreading programming as it relates to our project. The key terms used for multithreading in this project have the following meanings:

Definition 3.8: A **Thread** is a sequence of instructions executed within the context of a process. The term is used to refer to a separate path of execution for code.

Definition 3.9: The term **process** here is used to refer to a running BPEL instance, which can encompass threads.

Definition 3.10: The term **task** is used to refer to the abstract concept of work that needs to be performed.

Definition 3.11: Concurrency exists when at least two threads are in progress at the same time.

Definition 3.12: Parallelism arises when at least two threads are executing simultaneously.

Threads are one of several technologies that make it possible to execute multiple code paths concurrently inside a single application. They are a relatively lightweight way to implement multiple paths of execution inside of an application. At the system level, programs run side by side, with the system doling out execution time to each program based on its needs and the needs of other programs. Inside each program however exists one or more threads of execution, which can be used to perform different tasks simultaneously or in a nearly simultaneous manner.

Multithreading is a concept which is related to process and thread concept. A process may be divided into a number of independent units known as *threads.* A *thread* is a dispatchable unit of work. Threads are light-weight processes within a process. A process is a collection of one or more threads and associated system resources. The difference between a process and a thread is that a process may have a number of threads in it. A thread may be assumed as a subset of a process.

Multitasking of two or more processes is known as *process-based-multitasking* and is not in the scope of this project. In fact, process-based-multitasking is totally controlled by the operating system but thread-based multitasking can be controlled by the programmer to some extent in a program. In the context of this project, multithreading concept is used to allow the definition of branches within a BPEL process to run concurrently.

CHAPTER FOUR
DESIGN AND IMPLEMENTATION OF xBPEL

4.0. Introduction

This chapter discusses the technical design and implementation of an extension of BPEL: xBPEL. xBPEL consists of two major components: the testing framework and the logic support.

- **Conceptual framework:** The xBPEL implements the actual testing framework by using partner simulation, invocations of the process from Web Services Provider and result gathering.
- **Logic support:** The xBPEL extension also consists of utilities. These utilities define each unsupported pattern as extension of an abstract class Activity.

In section 4.1, the basic design considerations for the framework and logic support are presented on a high level that is using diagram class. These are used for the concrete software design in later sections. Section 4.2 discusses the implementation of the xBPEL – the prototype implementation for each extension points.

4.1 Design of xBPEL

Our review of related works on BPEL allowed us to identify more than seven control flow patterns not supported by WS-BPEL 2.0. However, our proposed extension only covers the five control flows described in section 2.1.3. That is because the two other control flow patterns herein Multiple Instance (MI) with a Priori Runtime Knowledge and MI without a Priori Runtime Knowledge require setting up an environment for testing and they depend on the type of BPEL engine used. In this section, the design to achieve the implementation of xBPEL is presented. The design is based on class diagram description where each unsupported control flow represents a class.

This section gives details on the design i.e. diagram classes (UML diagram) used in the next chapter. This shows only the abstract class Activity and the xBPEL extension proposed in this project i.e. each class for each control flow pattern. Figure 4.1 shows the diagram class used for xBPEL implementation.

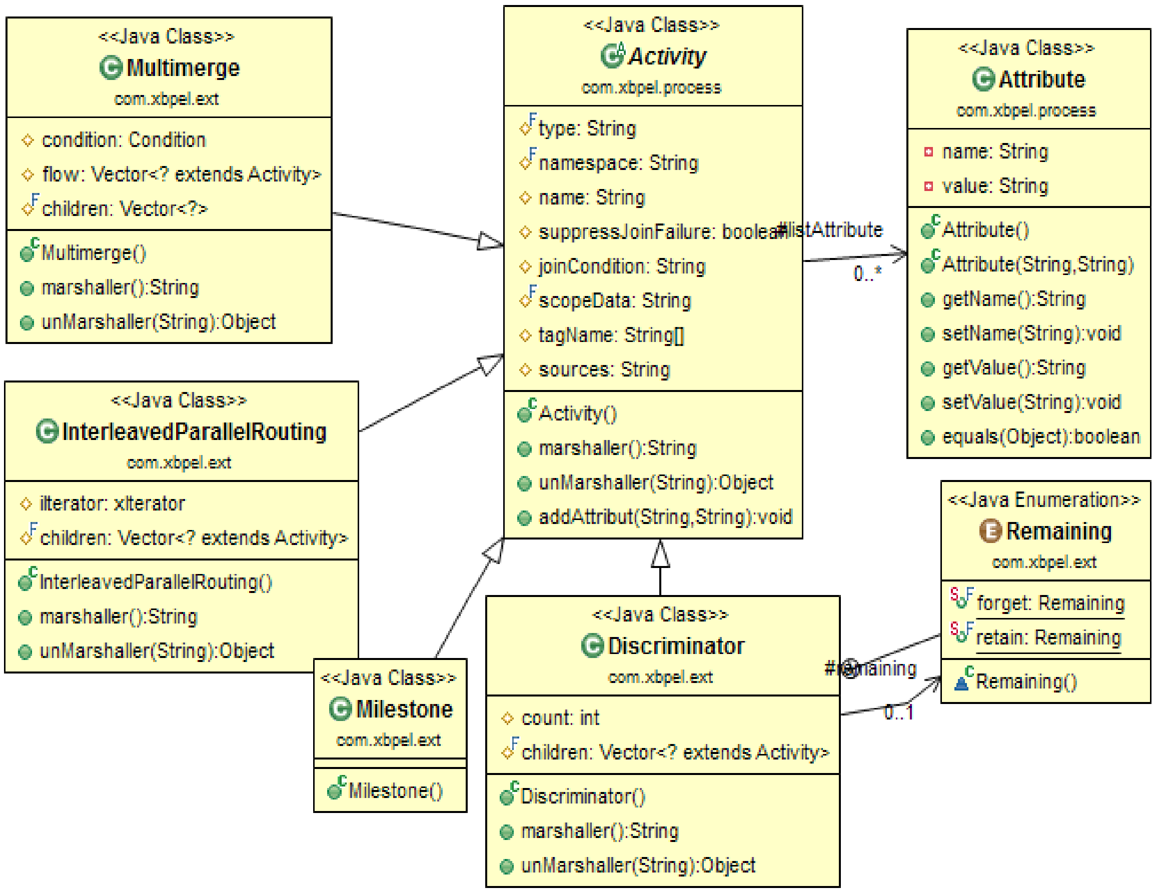


Figure 4.1: xBPEL diagram class

The BPEL reader is a utility which is responsible for reading BPEL files (.bpel format) and building up its object representation. A BPEL engine by default does not know how to create an instance of our extended activities. Therefore, a custom deserializer and serializer must be written. A deserializer allows the reading of a passed XML document while a serializer allows the writing to an XML file. In the context of this project, the deserializer and the serializer were achieved through the use of Unmarshaller and Marshaller classes defined in JAXB library. These two methods are defined in the abstract class Activity to allow our extension to be built up into its objects representation and vise-versa. Listing 4.1 shows the unmarshaller (reader) method which creates a new instance of an extended activity from the passed in XML files.

```java
public Object unMarshaller(String filename){
        try{
        JAXBContext context = JAXBContext.newInstance(Activity.class);
        Unmarshaller unMarshal = context.createUnmarshaller();
        Object obj = (Object)unMarshal.unmarshal(
                          new StreamSource(new StringReader(filename)));
                return obj;
        }catch(Exception e){
                e.printStackTrace();
                return null;
        }
}
```

Listing 4.1: Unmarshaller method (deserializer)

As BPEL reader deserializes an XML file into its object representation while the BPEL writer serializes the object representation to an XML file. We created a marshaller (writer) based on Marshaller class also defined in the JAXB library. Listing 4.2 shows the marshaller method such as defined in the abstract class Activity.

```java
public String marshaller(){
JAXBContext context = null;
        try {
                context = JAXBContext.newInstance(Activity.class);
                Marshaller marshal = context.createMarshaller();
                marshal.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
Boolean.TRUE);
                ByteArrayOutputStream output = new ByteArrayOutputStream();
                marshal.marshal(this, output);
                return output.toString();
        } catch (JAXBException e) {
                e.printStackTrace();
        }
        return null;
}
```

Listing 4.2: Marshaller method (serializer)

Since the two methods are defined in the abstract class Activity and that our extension classes inherit from Activity class, it automatically makes the two methods available to the child class's scope. This means, a class like Discriminator for example can be written to a XML file and the latter can be read back to get the object representation of the discriminator pattern.

The next section focuses on the explanation of the extension as well as the prototype's implementation. In fact, details on how to use the aforementioned design were shown, especially in the achieving of some prototype's implementation. In addition to that, tests on the prototype were performed to give proof of the extension's feasibility.

4.2 xBPEL and prototype implementations

The WS-BPEL specification requires using an <*extension*> element in a process definition to signal the presence of an extension. This declaration alone does not change the behavior of a WS-BPEL process activity (Allen 2008). It simply advertises the extension namespace; it is the use of attributes or elements belonging to the extension's namespace within a WS-BPEL construct that changes the process' or activity's behavior. Our process definition indicates its uses through the following declaration.

```
<bpel:extensions>
     …………
     <bpel:extension mustUnderstand = "yes"
                namespace = "http://www.xbpel.org/cs/bpelextension"
</bpel:extensions>
```

Listing 4.3: Extension Validation

The *mustUnderstand* attribute in the WS-BPEL specification allows a BPEL engine in dealing with extensions. Based on the value of this attribute, the engine has the option of rejecting the process definition, accepting it but ignoring any usages of the extension or accepting it and applying the rules associated with the extension.

As stated in the methodology, the extension provided in this project uses the notation xbpel to distinguish the proposed constructs from the standard BPEL constructs. That is, all other constructs used without xbpel notations are part of the BPEL specification.

4.2.1 Multi-merge pattern

As the matter of recall, a multi-merge pattern allows each incoming branch to continue independently of the others, enabling multiple threads of execution through the remainder of the process. For example, when concurrent activities A and B are multi-merged into C, the possible combinations of execution are: ABCC, ACBC, BACC, and BCAC. This pattern is known as uncontrolled join in (Michael 2005). Multi merge facilitates common logic for branches upon completion in process that permits simultaneous auditing and processing of an application.

The main reason why BPEL does not support multi-merge pattern is because the language is block structured-based and does not allow for two threads of execution to run through the same path in a single process instance. In fact, most other languages such as BPEL and Business process Modelling Language (BPML) are stumped by this pattern because they do not allow the same activity to be executed by separate threads (Michael 2005). Therefore, any extension aiming at providing support for multi-merge pattern must ensure that no more than one thread runs a single instance. To overcome the above BPEL's limitation and constraint, we proposed the multi-merge activity as a combination of two logical operations: the OR and JOIN operations or OR-split-and-JOIN in short. In fact, the multi-merge pattern can be split up into two subsequent activities: a multiple execution of subactivities herein represented by the prefix *multi* and a merging subactivity herein represented by the suffix *merge*. Consequently, we proposed the OR operation to model the subsequent activity *multi* and JOIN to model the *merge* activity. Each of these operations is defined as follows:

The OR operation defined in this extension for multi-merge pattern allows two or more branches to run concurrently. In fact, the OR operation guarantees that, at any point in time, there is at least one running branch if not all. During this stage, parallel execution of multiple branches can occurs. WS-BPEL provides support for parallel execution by the use of <*flow*> construct (Table 2.3). That allows to achieving the first subsequent activity of a multi-merge pattern. However; the difficulties come when performing the JOIN operation. The JOIN operation will basically allow the two or more branches, here defined by threads, to be joined into a single path or instance causing the execution of multiple threads over the single path. However, neither XLANG, nor WSFL – BPEL's forerunners – allow for two or more active threads following the same path. To cross this problem, the said extension suggests that the *merge* activity must be created as a result of an independent thread from existing threads. Listing 4.4 shows the proposed construction for multi-merge pattern.

```
<xbpel:multimerge>*
     <bpel:condition>. . .</bpel:condition>?
     <!-- To ensure concurrency for activities -->
     <bpel:flow>+
         <bpel:activityA conditionSatified = "yes | no"? />*
               . . . .
         <bpel:activityB conditionSatified = "yes | no"? />*
               . . . .
         <bpel:activityC conditionSatified = "yes | no"? />*
               . . . .
     </bpel:flow>
</xbpel:multimerge>
```

Listing 4.4: Multi-merge construct

As shown in Listing 4.4, each activity must be defined with an attribute *conditionSatisfied* whose value is either *yes* or *no*. In addition to this, a multi-merge construct must define an obligatory child activity <*condition*> which refers to the normal <*condition*> activity in BPEL specification and has the same behavior as defined in the standard. In fact, activities must run concurrently to satisfy the multi principle and then proceed the merge activity. The merging activity must be executed once for each incoming activity. However, when an activity satisfies the condition to get into the merging activity, it must first ensure that no other activity is actually using the merging activity. This can be done by defining a shared resource which is automatically locked by the activity running the merge path. Thus, each incoming activity must check the availability of that resource before undertaking that path. The following prototype implementation shows how the proposed extension can be achieved.

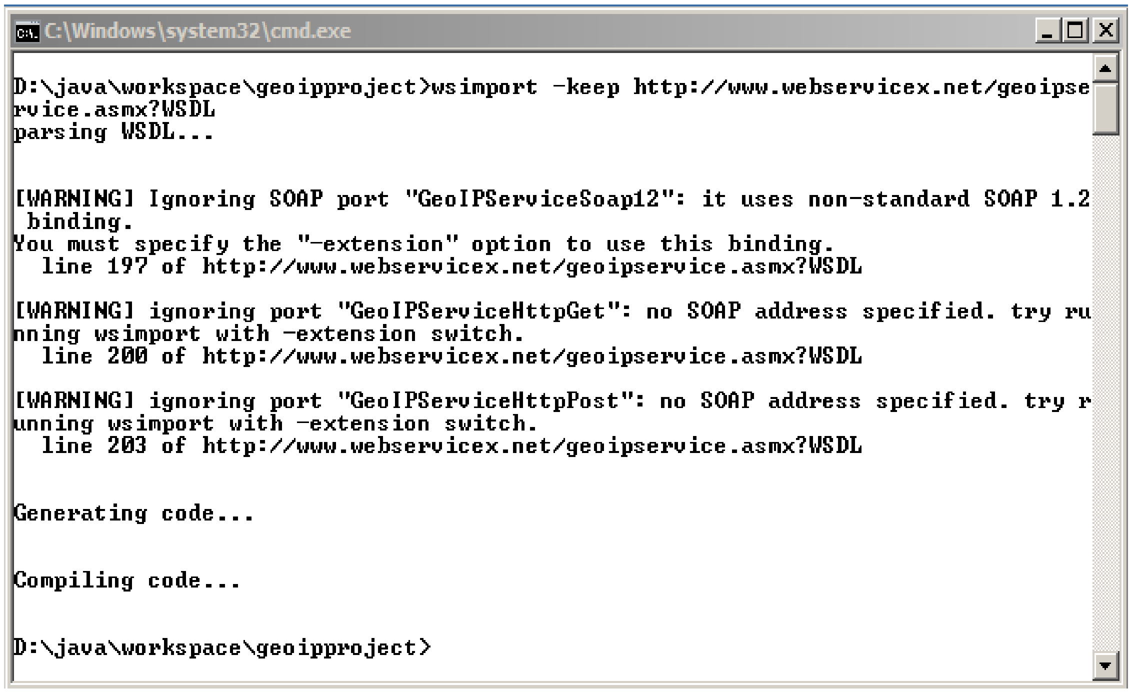
The example given in section 2.2.3 concerning multi-merge pattern was shaped around a recruitment process which consists of checking the references of the future employee by contacting two different services providers (professional references and personal references) and notifying the Human Resources Manager (HRM) each time a reference is checked and approved. Consequently, a prototype should use this description to implement the multi-merge pattern proposed in this project. However, due to unavailability of publicly accessible web services for references checking, alternative option was used. We defined a new example which consists of accessing two databases for Geo IP Service over Internet (similar to the professional and personal references) and get the cities for that given location – a country – (similar to the HRM's notification).

Indeed, a Geo IP Service enables organization to look for country by IP address. The prototype uses three publicly accessible Web Service Providers.

The first Web Service Provider used to implement the proposed multi-merge construct is WebserviceX.net. WebserviceX.net is a publicly accessible Web Service Provider available at 173.201.44.188 IP address. WebserviceX.net provides programmable business logic components and standing data that serve as “black boxes” to provide access to functionality and data via web services. The WebserviceX.net is located in United Sates, region of Arizona, latitude and longitude of 33.6119 – 111.8906. This Web Service Provider provides many others web services among which are the Geo IP service. The WSDL file that describes the service is available at http://www.webservicex.net/geoipservice.asmx?WSDL and supports the following operations:

- GetGeoIP operation that enables organization to look up countries by IP addresses
- GetGeoIPContext operation that enables organization to easily look up countries by Context.

The *wsimport* command was used in the command line to generate the necessary artifacts for the implementation of the prototype. Figure 4.2 shows the execution of the *wsimport* command for WebserviceX.net Web Service Provider.



```
C:\Windows\system32\cmd.exe

D:\java\workspace\geoipproject>wsimport -keep http://www.webservicex.net/geoipse
rvice.asmx?WSDL
parsing WSDL...


[WARNING] Ignoring SOAP port "GeoIPServiceSoap12": it uses non-standard SOAP 1.2
 binding.
You must specify the "-extension" option to use this binding.
  line 197 of http://www.webservicex.net/geoipservice.asmx?WSDL

[WARNING] ignoring port "GeoIPServiceHttpGet": no SOAP address specified. try ru
nning wsimport with -extension switch.
  line 200 of http://www.webservicex.net/geoipservice.asmx?WSDL

[WARNING] ignoring port "GeoIPServiceHttpPost": no SOAP address specified. try r
unning wsimport with -extension switch.
  line 203 of http://www.webservicex.net/geoipservice.asmx?WSDL


Generating code...


Compiling code...


D:\java\workspace\geoipproject>
```

Figure 4.2: wsimport command for webservicex artifacts

As it can be observed in Figure 4.2, the code was generated and compiled successfully though three warnings were found. In fact, the [WARNINGs] in lines 197, 200 and 203 are ignored because the binding defined for these ports uses non-standard SOAP 1.2 binding (see section 2.1.1.3 – Binding element). Moreover, the operation being affected here are *GeoIPServiceHttpPost* and not GetGeoIP which is the service utilized for this project. In fact, for the prototype implementation, only GetGeoIP operation was used and is defined as follows *public GeoIP GetGeoIP(String ipaddress).* The expected content of *GeoIP* contains the following schema fragment (see Listing 4.5) and the full implementation of the class GeoIP.java can be found in Appendix A.

```
<complexType name="GeoIP">
   <complexContent>
     <restriction base="{http://www.w3.org/2001/XMLSchema}anyType">
       <sequence>
         <element name="ReturnCode" type="xsd:int"/>
         <element name="IP" type="xsd:string" minOccurs="0"/>
         <element name="ReturnCodeDetails" type="xsd:string" minOccurs="0"/>
         <element name="CountryName" type="xsd:string" minOccurs="0"/>
         <element name="CountryCode" type="xsd:string" minOccurs="0"/>
       </sequence>
     </restriction>
   </complexContent>
 </complexType>
```

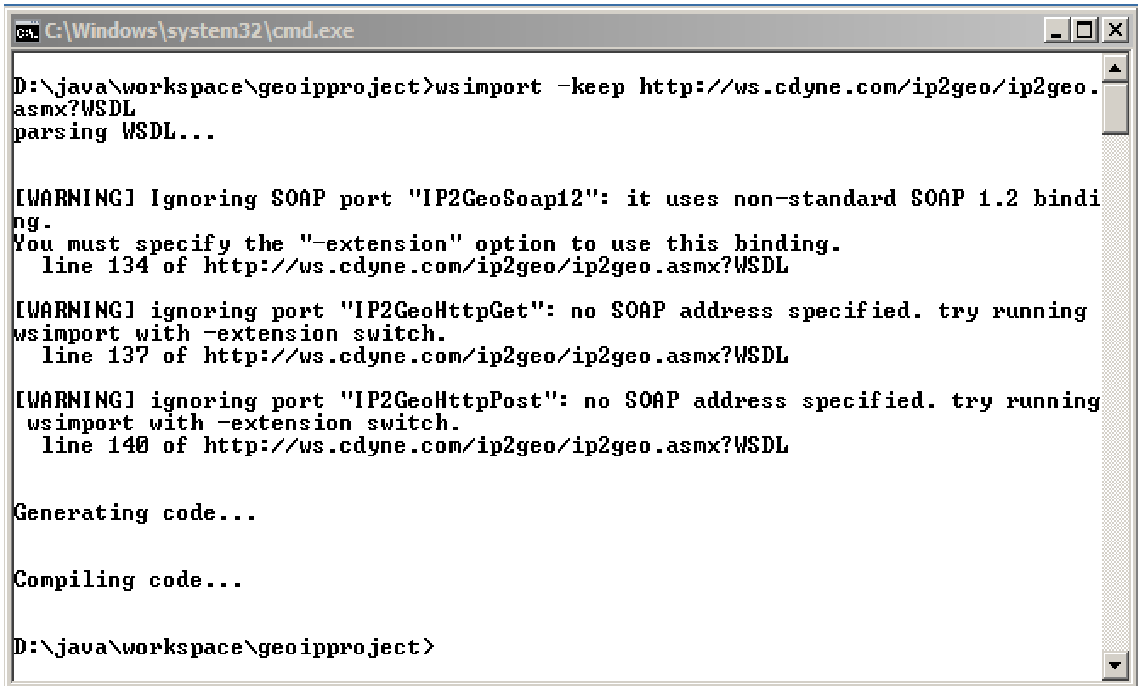
Listing 4.5: GeoIP schema fragment

The second Web Service Provider used to assess multi-merge pattern was ws.cdyne.com accessible at 4.59.146.110 IP address and located at United States, latitude and longitude 38 – 97. Ws.cdyne.com provides many others web services but the service used for multi-merge pattern was the Geo IP Service. The WSDL file that describes that service is also available at http://ws.cdyne.com/ip2geo/ip2geo.asmx?WSDL and supports only one operation defined as follows *public IPInformation resolve(String ipaddress, String licenceKey).* The expected content of *IPInformation* contains the following schema fragment (see Listing 4.6) and the full implementation of the class IPInformation.java can be found in Appendix A.

```
<complexType name="IPInformation">
   <complexContent>
     <restriction base="{http://www.w3.org/2001/XMLSchema}anyType">
       <sequence>
         <element name="City" type = "xsd:string" minOccurs="0"/>
         <element name="StateProvince" type="xsd:string" minOccurs="0"/>
         <element name="Country" type="xsd:string" minOccurs="0"/>
         <element name="Organization" type="xsd :sring" minOccurs="0"/>
         <element name="Latitude" type="xsd:double"/>
         <element name="Longitude" type="xsd:double"/>
         <element name="AreaCode" type="xsd:string" minOccurs="0"/>
         <element name="TimeZone" type="xsd:string" minOccurs="0"/>
         <element name="HasDaylightSavings" type="xsd:boolean"/>
         <element name="Certainty" type="w3.org/2001/XMLSchema}short"/>
         <element name="RegionName" type="xsd:string" minOccurs="0"/>
         <element name="CountryCode" type="xsd:string" minOccurs="0"/>
       </sequence>
     </restriction>
   </complexContent>
 </complexType>
```

Listing 4.6: IPInformation schema fragment

Similarly to WebServicex.net, we used the *wsimport* command to generate the necessary artifact for the prototype implementation. Figure 4.3 shows a snapshot of the command line prompt.



```
D:\java\workspace\geoipproject>wsimport -keep http://ws.cdyne.com/ip2geo/ip2geo.
asmx?WSDL
parsing WSDL...


[WARNING] Ignoring SOAP port "IP2GeoSoap12": it uses non-standard SOAP 1.2 bindi
ng.
You must specify the "-extension" option to use this binding.
  line 134 of http://ws.cdyne.com/ip2geo/ip2geo.asmx?WSDL

[WARNING] ignoring port "IP2GeoHttpGet": no SOAP address specified. try running
wsimport with -extension switch.
  line 137 of http://ws.cdyne.com/ip2geo/ip2geo.asmx?WSDL

[WARNING] ignoring port "IP2GeoHttpPost": no SOAP address specified. try running
 wsimport with -extension switch.
  line 140 of http://ws.cdyne.com/ip2geo/ip2geo.asmx?WSDL


Generating code...


Compiling code...


D:\java\workspace\geoipproject>
```

Figure 4.3: wsimport command for cdyne service artifacts

Here also, [WARNINGs] in lines 134, 137 and 140 are being ignored because the binding defined for these ports uses non-standard SOAP 1.2 binding (see section 2.1.1.3 – Binding element) and this does not affect the testing of the prototype implementation since the operation being used is *resolve*.

The third Web Service Provider provides weather forecast and region information for a given place name. It is available in www.restfulwebservices.net at 184.168.27.33 IP address, United States, Arizona region, latitude and Longitude of 33.6119 – 111.8906. The WSDL document that describes the services is available at http://www.restfulwebservices.net/wcf/ WeatherForecastService.svc?wsdl. The service requested from this Provider was Get cities names and has the following header *public ArrayOfstring getCitiesByCountry(String country)*. This function returns a list of string containing the names of cities for a given country. The expected result is shown in Listing 4.7

```
<complexType name="ArrayOfstring">
   <complexContent>
     <restriction base="xsd:anyType">
       <sequence>
                <element name="string" type="xsd:string"
       maxOccurs="unbounded" inOccurs="0"/>
       </sequence>
     </restriction>
   </complexContent>
 </complexType>
```

Listing 4.7: ArrayOfstring schema fragment

Figure 4.4 shows a graphical representation of the location process described above.

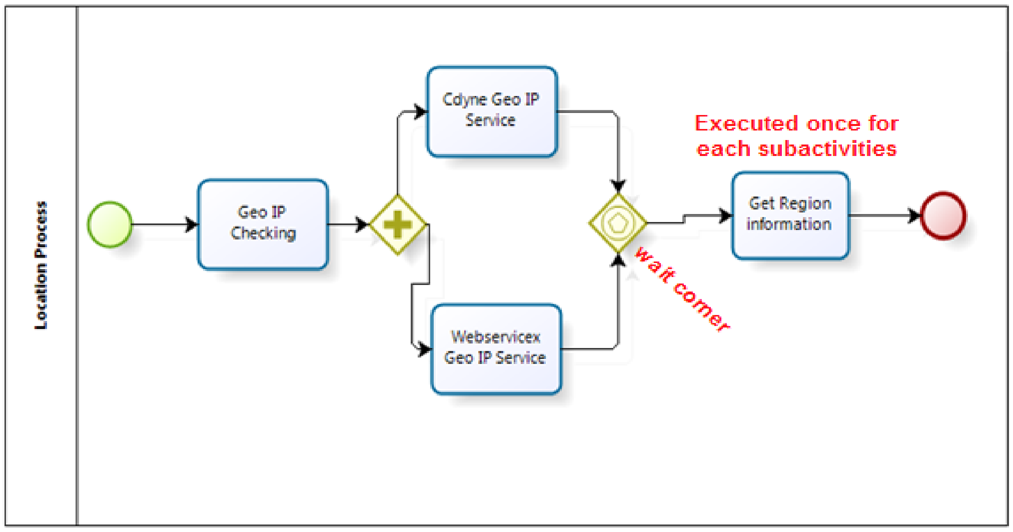


Figure 4.4:Multi-merge pattern: Region Information

From the above scenario, the prototype consisted of accessing the name of a place herein the country name, and passing this name to the Get Region information Service which in turn provides us with a list containing the name of cities for this country. When one of the Geo IP Service replies, Get Region information must be executed. Therefore, to achieve this scenario, we defined two subactivities characteristics of multi-merge pattern (Multi and Merge activities). The multi subactivity was implemented as the result of two concurrently activities, two classes (Webservicex.java and Cdyne.java) were implemented. This behavior is allowed by BPEL and can be achieved through the use of *<flow>* construct. However, after the multi subactivity, the flow for the multi-merge pattern does not proceed with the same concept of multiple instances, but merge the multiples instances into one instance which in our scenario is achieved at the *wait corner* (see Figure 4.4). To mimic this behavior our prototype implemented a Get Region activity after the two Geo IP Services which execution is based on the reply of the two prior services. In fact, Get Region information must be executed once for each incoming replies. However, leaving it such as defined might cause the get cities activity to be executed simultaneously by the Geo IP Services if replies come simultaneously. To avoid this, for we know it is not allowed in BPEL, we proposed that a static variable herein defined as *public static final int waitSignal* which default value is 1, was used. This variable aims at simulating the shared resource or the lock/unlock behavior aforementioned. In fact, when one of the processes runs through the get cities activities, it must set *waitSignal* to 0 (lock state) and make it not available for other incoming processes and when it finishes, it must set waitSignal to 1 (unlock state). That is, an activity can only start the get cities activity when *waitSignal* variable or the shared resource is freed, by so doing; we ensure that not more than two threads execute the merge activity. This allows our prototype to conform to BPEL constraint for not allowing more than two threads should run through the same path. Listing 4.8 shows the implementation of the multi-merge class which can be used to marshal (write) and unmarshal (read) and .bpel files into or from a its object representation.

```
@XmlRootElement(name = "multimerge",  namespace =
"http://www.xbpel.org/cs/bpelextension")
public class Multimerge extends Activity{
        @XmlElement
        protected Condition condition;
        /**
         * Flow activity
         */
        @XmlElement
        protected Vector<? extends Activity> flow;
        /**
         * A list of all the child activities.
         */
        @XmlElement
        protected final Vector<?> children = new Vector<Object>();
        @Override
        public String marshaller() {
                return super.marshaller();
        }
        @Override
        public Object unMarshaller(String filename) {
                return super.unMarshaller(filename);
        }
}
```

Listing 4.8: multimerge class

4.2.2 Discriminator pattern

The second control flow pattern to be considered in this section is discriminator pattern which refers to a point in the process that waits for one of the incoming branches to complete before activating the subsequent activity (see section 2.1.3). In this project, we implemented an extension of BPEL to support discriminator pattern as an OR-split-XOR-join condition.

In the discriminator pattern, when multiple parallel branches converge at a given join point, exactly one of the branches is allowed to continue on in the process, based on a condition evaluated at runtime, the remaining branches are blocked. It is a special case of the N-out-of-M join pattern where M parallel branches meet at a point of convergence and only the first N are let through.

The reason why BPEL offers no support for discriminator pattern is the same as in multi-merge pattern. In fact, the BPEL specification does not allow a second instance of an activity to be created if the first instance is still active. This does not provide a solution for the discriminator. The solution provided in our extension consists of finishing the first instance before creating any second instance and implements the OR-split-XOR-join condition as follows.

The OR-split stage defined in discriminator construct has the same behavior as the one defined in multi-merge pattern and allows the two branches to run concurrently through the use of <*flow*> tag as defined in BPEL specification (Table 2.2).

The XOR-join operation herein defined ensures that only one thread continues the flow after the concurrent execution. In fact, the XOR is an exclusive OR operation which is true for the execution of only one instance. What it means is that, for two logical predicates A and B, the XOR operation returns true only when A is true, or B is true, but for A and B true, the reply is false. Briefly, it ensures exclusivity which means rejecting the execution when two elements share the same state. It is that behavior that is used for the discriminator pattern herein defined. In fact, after the OR-split execution which allows concurrent execution, only one branch should pursue the execution. This exclusion is achieved through the use of XOR-join condition within the flow.

To implement the aforementioned logic for the discriminator pattern in BPEL, we defined a new construct called discriminator with a new set of attributes *remaining* and *count*.

The *remaining* attribute which value is either '*forget*' or '*retain*' tells the BPEL engine either to '*forget*' the other branches or to *retain* them. What it means is that, the BPEL engine can either terminate the other branches and only continues the execution with one branch or keep remaining branches for subsequent usages. However, the default behavior of a discriminator pattern is to forget and terminate the other branches. A discriminator construct with an attribute *remaining* set to '*retain*' will be suited in a situation whereby the subsequent processes will be later invoked in the business process. And any other situations whereby the process will make no subsequent calls to the partner links in the nearest future can set the *remaining* attribute to '*forget*'. This latter option will ensure that the BPEL engine closes the other branches except the chosen one.

The *count* attribute which value is in the range of integer value, tells the BPEL engine the number of activities, participants or links that need to reply first before the flow will continue with the next branch. The default value of this attribute is 1. In fact, by default, only one branch is expected before pursuing the flow and the other branches forgotten. Other suggestions on how to terminate a process can be achieved through the use of *close* construct defined in BPEL specification (OASIS - BPEL Technical Committee, 2007). Listing 4.9 shows the proposed extension for discriminator pattern.

```
<xbpel:discriminator
        remaining = 'forget | retain'?
        count = 'xsd:int'?
>*
     <bpel:ActivityA>
        //define here the flow for an activity
        // can be an activity encapsulated within a sequence construct
        // or a flow construct
     </bpel:ActivityA>
     <bpel:ActivityB>
        //define here the flow for another activity
        //a discriminator pattern needs a minimum of two activities in
        // order to proceed
     </bpel:ActivityB>
</xbpel:discriminator>
```

Listing 4.9: Discriminator construct

The above construct was applied to the example first described in section 2.1.3 which consists of improving query response time, by sending a complex search to two different databases over the Internet, the first one that comes up with the result proceed, the second one is ignored. The prototype for discriminator pattern implemented for this scenario uses the two Web Service Providers first defined for the multi-merge pattern which are Cdyne and Webservicex. In fact, the two WSDL describing the two web services were retrieved from their repository and the required classes for implementing the client side business logic was generated using *wsimport* tool. Afterward, the OR-split operation was implemented by defining two processes carried by two threads. This has allowed the two processes to open connections to the web services concurrently. Then a *while (true)* statement was used to simulate the wait condition and a *break* command invoked when receiving result from one of the opened connection. Consequently, when one of the two concurrent connections or threads replies, the other thread is 'forgotten'. Also, to ensure that only one thread of execution runs through the path, the two remaining threads – main thread and the unforgotten thread – are joined into one (JOIN operation) and the flow continues. Figure 4.5 shows the scenario described for discriminator pattern.

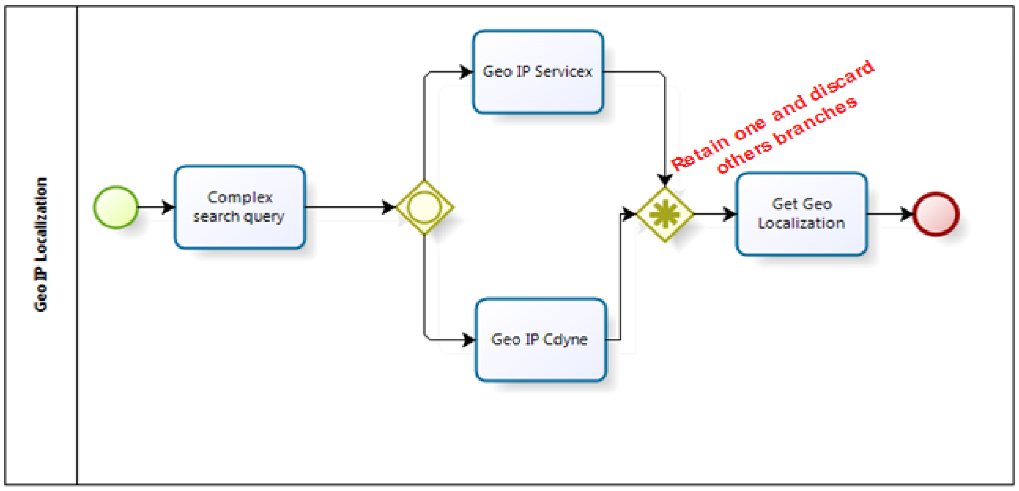


Figure 4.5: Discriminator pattern for the Geo IP localization scenario

A full implementation of this scenario can be obtained in the Appendix A. Using destinations IP addresses of 15.15.15.15, 189.12.15.78 and 214.12.211.18, the Geo IP

Localization scenario was executed. Table 4.1 shows the response time in milliseconds (msecs) for each IP address destinations. In fact, depending on some factors such as the throughput, delay and the load of the link at that moment, the connection to the Service Provider Cdyne can be slower than the connection to the Service Provider Webservicex. Discriminator pattern would query the two databases represented by Cdyne and Webservicex and only keep the response from the first Provider to reply.

Table 4.1 shows a summary of repeated execution of discriminator pattern for IP Geo localization.

Time	Destination	Location	Response time (msecs.)	WS Providers
16/04/2014				
3:40pm	15.15.15.15	United States	1060	Cdyne
3:50pm	189.12.15.78	Brazil	1006	Webservicex
4:00pm	189.12.15.78	Brazil	5012	Cdyne
5:30pm	214.12.211.18	United States	1540	Webservicex
20/04/2014				
12:20pm	214.12.211.18	United States	4680	Webservicex
12:30pm	214.12.211.18	United States	1116	Webservicex
12:40pm	214.12.211.18	United States	3212	Cdyne

Table 4.1: Sample of discriminator pattern execution

As shown in Table 4.1, the first set of execution carried on 16/04/2014 shows that the Provider Cdyne was the first to reply with a response time of 1060 msecs. 10 minutes later, at 3:50pm, Webservicex was the first database to reply with a response time of 1006 msecs. We can generally see that, depending on the condition of the network at that moment, we received reply from one Provider with different response time and we do forget the other branch (other Provider). An execution of the discriminator pattern executed on 20/04/2014 also shows that Webservicex Provider outperformed the Cdyne Provider for the first two replies.

Listing 4.10 shows the implementation of the discriminator class which can be used to marshal (write) and unmarshal (read) a .bpel files into or from its object representation.

```
@XmlRootElement(name = "discriminator",
         namespace = "http://www.xbpel.org/cs/bpelextension")
public class Discriminator extends Activity{
         enum Remaining {
                  forget,
                  retain
         }
         @XmlAttribute
         protected Remaining remaining;
         /**
          * A list of all the child activities.
          */
         @XmlElement
         final Vector<?> children = new Vector<Object>();
         @Override
         public String marshaller() {
                  return super.marshaller();
         }
         @Override
         public Object unMarshaller(String filename) {
                  return super.unMarshaller(filename);
         }

}
```

Listing 4.10: Discriminator pattern prototype

4.2.3 Arbitrary cycles pattern

As a matter of recall, arbitrary cycles refer to a point in a workflow process where one or more activities can be done repeatedly (section 2.1.3). The arbitrary cycles is not supported by WS-BPEL specification because of the restriction made that links cannot cross the boundaries of a loop and whose links may not create a cycles. Thus, extension to support this pattern must conform to these restrictions and follow the rule. The arbitrary cycle pattern allows part of a process to jump back to arbitrary parts and can have more than one entry point and belong to a set of pattern known as cycle. These sets of pattern are aimed at depicting repetitive behaviour in a business process model (e.g. arbitrary cycle, structured loop). A structured cycles which has one entry point and one exit point can be achieved in BPEL through the use of *while* activity and need no extension of the BPEL 2.0.

Sometimes, arbitrary cycles can be converted into structured cycles and these structured cycles can then be mapped directly onto BPEL "*while*" activities. However, not all non-structured cycles can be converted into structured ones when AND-splits and

AND-joins are involved. In addition, (Kiepuszewski, ter Hofstede, & Bussler, 2000) and (Kiepuszewski, 2003) showed that there are situations where it is possible to transform process model containing arbitrary cycles into structured processes, thus allowing them to be captured in offerings based on structured process models like BPEL, however, not all unstructured processes that can be transformed into structured processes and these can be seen as mere work-around solution rather than full support of the pattern.

In fact, arbitrary cycle pattern provides a means of supporting looping that is either unstructured or is not block-structured and it must also be possible for individual entry and exit point to be associated with distinct branches. There are actually no specific context conditions associated with this pattern and BPEL is not able to represent arbitrary process structures because the language is block structured. Nevertheless, in Business Process Model and Notation (BPMN), it is possible to create an arbitrary cycle pattern by connecting the sequence flow to any upstream activity. Figure 4.6 shows a graphical representation of a process which contains an arbitrary cycle. It is based on the description provided in section 2.1.3 for the arbitrary cycle.

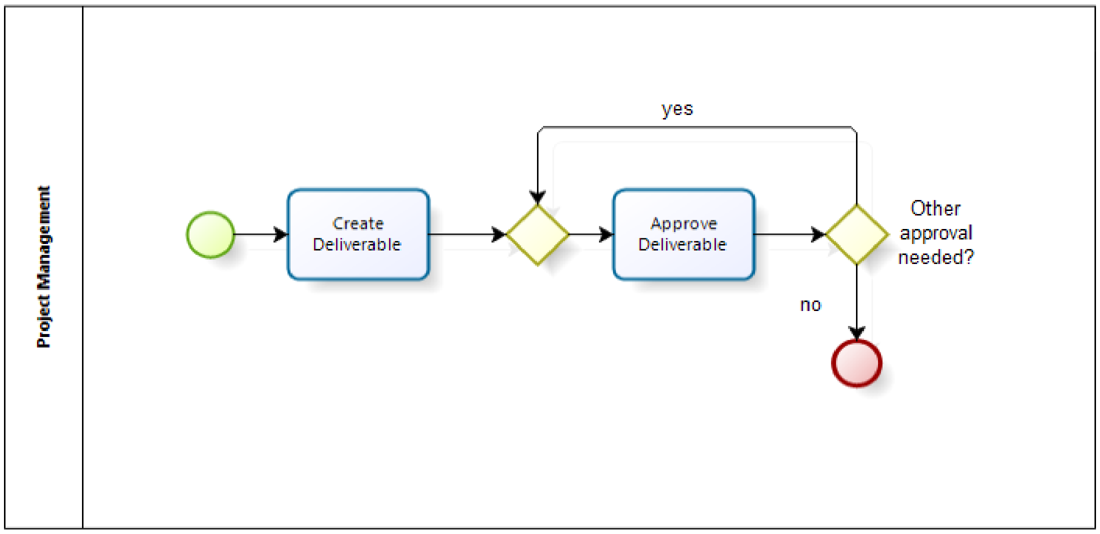


Figure 4.6: A representation of an arbitrary cycle

Proposing an extension of BPEL to support arbitrary cycles involves moving BPEL from a block structured language based on a non structured language. Such initiative will require changing BPEL specification at its root level and its underlying concepts. This makes the support for such pattern more than a mere extension which this work aims at providing. This fact allows the conclusion that, indeed, BPEL specification by no means can support arbitrary cycles. The reason for not supporting this pattern dwells not in the lack of gateways or logical operations, but in the way BPEL specification was conceived by its designers. In fact, what we are trying to prove here is that, providing support for arbitrary cycle implies changing the overall concept upon which BPEL is rooted. However, this is not the aim of this project and we aim not to be under such illusion. Nevertheless, our findings on BPEL specification state that, giving support for arbitrary cycle in BPEL can be achieved by making BPEL specification a dual-based block language which is block based language to support structured block as well as non block language for supporting unstructured block.

In a nutshell, BPEL specification would not support arbitrary cycles (unstructured blocks). The language being block structured, it cannot capture unstructured cycles unless support is provided for unstructured blocks.

4.2.4 Interleaved parallel routing pattern

Interleaved parallel routing is one of the control flow patterns where a work-around solution has been provided. (W.M.P, A.H.M, Kiepuszewski, & A.P., 2002) and (Petia, Wil M.P., & Marlon, 2002) also proposed work-around solutions which also have been discussed in section 2.2.2. In this section, we focus on providing context, construct and attributes to allow BPEL to support interleaved parallel routing from its root of its specification. As a matter of recall, this pattern occurs when a set of activities is executed in an arbitrary order and differs from arbitrary cycles. Each activity here is executed exactly once and the order is decided at runtime. The analysis of work-around solutions proposed by scholars for this pattern showed that complexity increases exponentially with the number of activities to be interleaved. In fact, the work-around solution consisted of defining all the possible interleave or permutation for a set of activities. For example, with three activities A, B and C; ABC, ACB, BAC, BCA, CAB and CBA represent all the possible sequence of execution of A, B and C for the interleaved pattern. This shows that, for *n* activities, the resulting complexity is *n!* (E.g. if $n = 9$, we have 9! = 362 880 activities to interleaved). This become more disgusting when *n* is big. Therefore, the proposed solution of interleaving all the sequence of activities, though reasonable for a few set of activities is not suited when *n* tends to infinity. Based on this observation, a proposed solution must consider the fact that complexity should not be affected by the number of activities. If not, this would lead to worst complexity case. In addition, another work-around and very simply, but unsatisfactory solution is to fix the order of execution i.e. instead of using parallel routing, sequential routing is used. Since the activities can be executed in an arbitrary order, a solution using a predefined order may be acceptable (section 2.2.2). However, by fixing the order, flexibility is reduced and the resources cannot be utilized to their full potential. Thus, we proposed an interleaved construct for supporting interleaved parallel routing based on XOR-split gateway. Choosing only one activity to be executed at the time satisfies the condition of exclusive decision.

The first step in achieving the interleaved parallel routing (IPR) pattern in our xBPEL is by defining within the construct *<xbpel:interleaved>* a set of activities to be interleaved. These set of activities can only be defined using their reference which must be an unique identifier within the *<xbpel:interleaved>* bloc. These references are then used to achieve the XOR operation during the execution. In fact based on this set of activities, the eXclusive OR will allow one activity to be executed at the time. Since we know that, XOR allows for activities to be executed exclusively. Details on the operation of the XOR gateway can be found in section 3.5. Beside this non-concurrent execution of interleaved activities, we suggest the *split* attribute to be applied after each execution. This split operation has an objective of splitting up the list or set of activities into a new sets of activities. In practical, the new list would be equal to the former list minus the reference of the interleaved activity being executed. By doing so, we ensure that, no activity will be executed more than once. Only activities which references are in the list of activities to be executed will be executed. Those activities that are already executed will be seen removed from that list and added in the temporary list. We call it temporary list because; it will have no subsequent used within the *<interlveaved>* construct. This temporary list doesn't need to be defined in the construct since it's left to the engine to use any locations for temporary used and free the location as soon as the execution finishes. Listing 4.11 shows the proposed extension for supporting interleaved parallel routing pattern.

```
<xbpel:interleaved>*
        <xbpel:iterator
                references = “$(a, b, . . ., c)”?
                executed = “xsd:list”
        </xbpel:iterator>?
        <activityA ref = “a”>*
                . . .
        </activityA>

        <activityB ref = “b”>*
                . . .
        </activityB>*
        <!—Define all the activities to be interleaved here -->
        <activityC ref = “c”>*
                . . .
        </activityC>
</xbpel:interleaved>
```

Listing 4.11 : interleaved proposed construct

As we can see from the above construct, the interleaved pattern must have at least two subactivities to be interleaved. They would all be referenced into the *references* attribute defined in the interleaved construct. The first element to start can be anyone on this attributes. When performed, the reference would go back to the executed list and add itself as executed. The iterator continues running as long as all activities referenced have not been performed.

The decision on which activity should start first in this implementation is left to the flow engine; it decides randomly which participant receives the hand first to proceed. It is not a concurrent-iterator. Each element in the references attribute will have its turn in the execution process. The *executed* attribute is there to make sure that, all the activities in the interleaved construct will be executed. Once an activity is retrieved by using its reference in the *references* attributes, the flow proceeds for that activity. As the process performs, the reference of the activity is removed from the list of references and added in the list of executed activities: *executed* attribute. The interleaved construct would take care of executing only one of its child at a time. The execution order of execution is randomly chosen.

Listing 4.12 shows the JAXB class used for the interleaved parallel routing.

```
@XmlRootElement(name = "interleaved", namespace =
"http://www.xbpel.org/cs/bpelextension")
public class InterleavedParallelRouting extends Activity{
        class xIterator{
                @XmlAttribute
                protected List<String> references;
                @XmlAttribute
                protected List<String> executed;
        }
        @XmlElement
        protected xIterator iIterator;
        /**
         * A list of all the child activities.
         * Refer to list of activities to be interleaved
         */
        @XmlElement
        protected final Vector<? extends Activity> children = new Vector<>();

        /**
         * @see com.xbpel.process.Activity#marshaller()
         */
        @Override
        public String marshaller() {
                return super.marshaller();
        }
        /**
         * @see com.xbpel.process.Activity#unMarshaller(java.lang.String)
         */
        @Override
        public Object unMarshaller(String filename) {
                return super.unMarshaller(filename);
        }

}
```

Listing 4.12: Interleaved parallel routing class

To implement a prototype for the interleaved parallel routing pattern, we made use of the example described in section 2.1.3 which consists of executing at the end of each year and for each account two activities: *"add interest"* and *"charge credit card costs"*. As it can be noticed, these activities can be executed in any order, but since they both update the account, they cannot be executed at the same time. Here is the description of how the interleaved parallel routing pattern was used to implement the above example.

Both "*add interest*" and "*charge credit card costs*" were implemented as the result of two activities following the event "*end of year*". What it means is that, when the event "*end of year*" is triggered by the business manager, the two activities aforementioned are executed. Since the two activities can be executed in any order but not simultaneously, we use the function random to determine the reference of the activity to start with. In fact, as described above, the references of activities to be interleaved are stored in a list of activities. At the first iteration which is made possible by the <*iterator*> construct we randomly choice an activity within the list of referenced activities. Using the class Random defined in *java.util.Random*, we were able to return an index between 1 and the number of interleaved activities, this index is used to retrieve the activity at the position index using the get function defined in *java.util.List<E>* by *E get(int index)* where *E* refers to a generic type which needs to be defined when the object is instantiated. Once an activity is selected, we execute the activity until its completion. When the activity is complete, the activity's reference is removed from the *references* list and then added in a list of executed activities here called *executed*. This list stores the list of activities already executed. At the second iteration, the same procedure is repeated and this time, different activity is selected in the *references* list. Since, after a single iteration the activity being executed is removed from the pool of activities referenced, we ensure that no activity is executed more than once. In the context of this prototype implementation, only two iterations are required to run through all the referenced activities since we only have two activities to interleave. Figure 4.7 shows a graphical representation of the example described above.

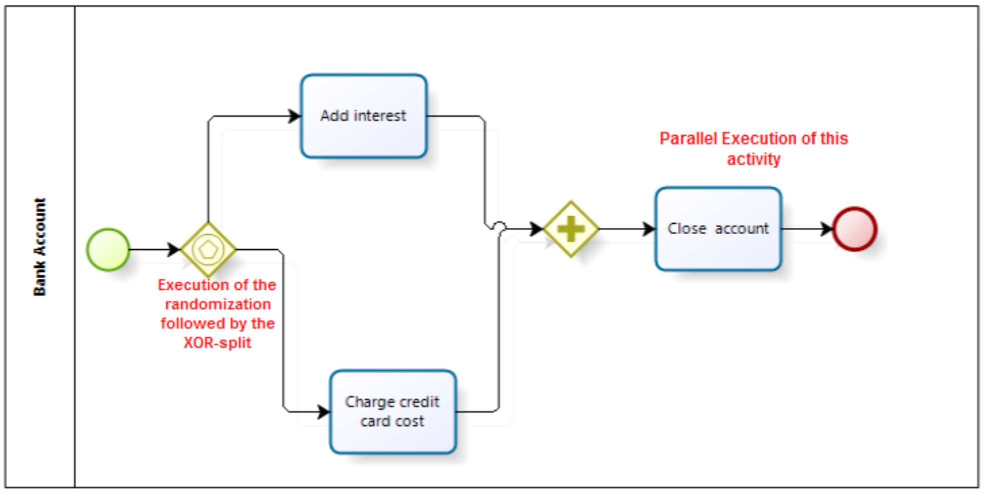
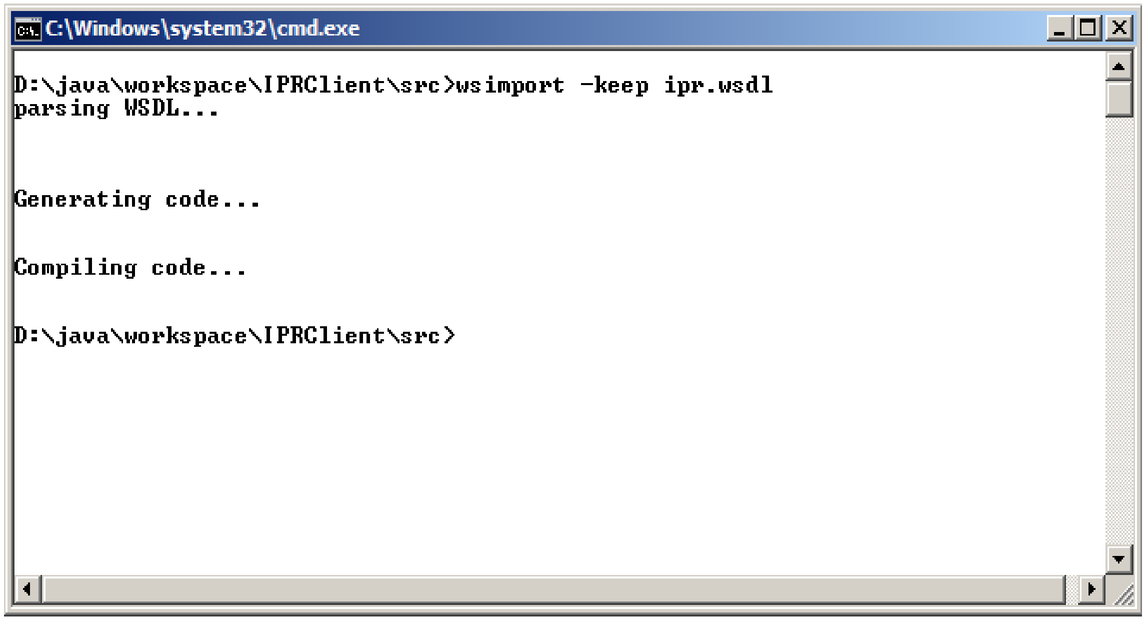

Bank Account
Add interest
Parallel Execution of this activity
Close account
Execution of the randomization followed by the XOR-split
Charge credit card cost

Figure 4.7: Interleaved parallel routing pattern example

To implement the above description, two Java classes *AddInterest.java* and *ChargeCredit.java* were defined. These two classes are meant to execute the "*add Interest"* and the *"charge credit card cost*" activities respectively. For this example, an internal Web Service Provider was used rather than an external. Artifacts to implement the client side were obtained by performing the *wsimport* command on a WSDL file. Figure 4.8 shows the result of the command.



```
D:\java\workspace\IPRClient\src>wsimport -keep ipr.wsdl
parsing WSDL...


Generating code...


Compiling code...


D:\java\workspace\IPRClient\src>
```

Figure 4.8: wsimport command account update artifacts

Code generated and compile successfully. The ipr.wsdl document can be found in Appendix A and can be used to generate the necessary artifacts for implementing the described scenario.

Using an internal Service Provider simply means that the WSDL file was not retrieved from any publicly available Web Service Provider but it was written and published to attain the objective of this project.

Table 4.2 shows four executions for the proposed interleaved construct. In fact, the two activities '*Add Interest'* and '*Charge credit card cost'* are executed exclusively and in any order. The first column (N°) numbers the execution while the second column (Order) shows which activity has started first by specifying the 1st or 2nd order. For example, for the first execution (N° = 1), it can be observed that, activity '*Add Interest'* started first with a time execution of 51 milliseconds (msecs) followed by *'charge credit cost'* with a time execution of 7 milliseconds. This makes a total execution time of 58 milliseconds. Also, for N° = 3 (third execution), *'charge credit cart cost'* was the first activity to start (138 msecs) and *'Add interest'* the second activity to start (12 msecs) which make a total execution time of 150 msecs.

The code used to generate Table 4.2 can be obtained by using *wsimport* utility in command line as shown in Figure 4.8

N°	Order	Add interest	Charge credit card cost	Total Execution Time
1	1st	**51 msecs**		**58 msecs**
	2nd		**7 msecs**	
2	1st	**109 msecs**		**127 msecs**
	2nd		**18 msecs**	
3	1st		**138 msecs**	**150 msecs**
	2nd	**12 msecs**		
4	1st		**122 msecs**	**132 msecs**
	2nd	**10 msecs**		

Table 4.2: Execution of the interleaved parallel routing pattern

As shown in the above table, the execution first started with the activity Add Interest which lasted for 51 milliseconds, followed by the Charge Credit card cost which lasted for 7 milliseconds. At the third execution, it can be seen that, random function returns the *"charge credit card cost"* reference first. This led to the execution of this activity which performed for 138 milliseconds followed by the "*add Interest"* for 12 milliseconds. This prototype also ends the description of the interleaved parallel routing pattern extension. Details about the generated artifacts can be found in the appendix A.

4.2.5 Milestone pattern

The last but not the least pattern to be considered is the milestone pattern. As a matter of recall, the milestone pattern defines a point in the workflow process where a determined milestone has to be reached to enable a given activity. That is why milestone pattern belongs to state based pattern category. In fact, the milestone pattern provides a mechanism for supporting the conditional execution of a task or sub-process (possibly on a repeated basis) where the process instance is in a given state. The notion of state is generally taken to mean that control-flow has reached a nominated point in the execution of the process instance (i.e. *milestone*). As such, it provides a means of synchronization two distinct branches of a process instance, such that one branch cannot proceed unless the other branch has reached a specified state.

The reason why BPEL does not provide direct support for milestone pattern is because the pattern is state-based. In fact, the necessity for an inherent notion of state within the process model caused this pattern not to be widely supported (W.M.P, Arthur, & Nick, Workflow Patterns Initiative, 2011).

In section 2.1.3 an example describing the milestone pattern was provided. It consisted of requesting approval for a request withdrawal. That is, an activity "*approve order withdrawal*" following an activity "*request withdrawal*". In this example, the milestone consisted of performing the "*approve order withdrawal*" if and only if (i) the activity "*place order*" is completed, and (ii) the activity "*ship order*" has not yet started. Figure 4.9 shows a graphical representation of the said example.

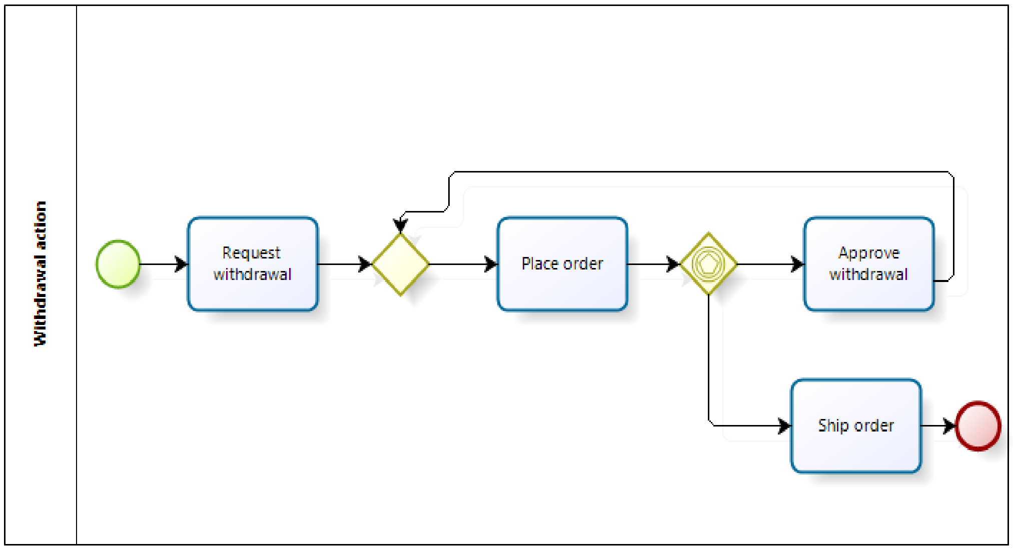


Figure 4.9: reversing a bank transaction example (milestone pattern)

For a milestone, it should be noted that the execution of the process containing a milestone phase does not influence the state of the milestone itself, i.e. unlike normal control flow dependencies it is a test rather than a trigger.

BPEL indeed does not provide full support for this pattern but this pattern can be indirectly achievable by using a *<pick>* construct within a *<while>* loop which can only be executed once but the solution is overly complex. This work-around solution was proposed by (W.M.P, A.H.M, Kiepuszewski, & A.P., 2002). It uses a deferred choice between executing the activity B, or executing activity E with a while loop to guarantee that as long as B is not chosen, E can be executed an arbitrary number of times (see section 2.2.2: Work-around solutions). Listing 4.13 shows the adopted work-around solution for our extension. In fact, as stated in the introduction of the chapter 2 on the methodology, control flow patterns are analyzed in order to identify the complex patterns which BPEL does not provide full support. Afterwards, existing work-around solutions for this pattern must be reviewed. If any work-around solution exists and is feasible, the solution is adopted. In the other hand, we proposed a new extension for that pattern.

For the all pattern thus far studied, the milestone pattern satisfied the last criteria, that explains why it was adopted and included as it is in the extension. In fact, with this, existing BPEL engines don't need to develop a new set of construct for supporting milestone pattern. Support is only achieved with the existing constructs (see section 2.1.2)

```
Activity A
B_completed := "false"
<while B_completed = "false">
  <pick>
   <onMessage me>
         Activity E
   </onMessage>
   <onMessage mb>
         <sequence>
                 B_complete := "true"
         </sequence>
  </pick>
</while>
Activity B
```

Listing 4.13: the adopted work-around solution for milestone (W.M.P, A.H.M, Kiepuszewski, & A.P., 2002)

Definition 4.1: BPELJ stands for Business Process Execution Language for Java

Beside this work-around solution proposed by (W.M.P, A.H.M, Kiepuszewski, & A.P., 2002) and adopted in xBPEL, a possible solution for implementing the milestone pattern is also provided by BPEJ and achieved through the use of the correlation element (Vasko & Dustdar, 2004).

4.3. Proposed xBPEL architecture

BPEL specification origin dates back to 2002 where the first specification BPEL4WS 1.0 was released followed by BPEL4WS 1.1 in 2003. From this time, BPEL4WS was submitted to OASIS for standardization which released a drafted version of BPEL 2.0 in 2004 and changed the name from BPEL4WS to WS-BPEL (See section 2.1.2.2). From there, many organizations started adopting WS-BPEL as their business language to automate their processes. However, BPEL engines that can achieve good performance and can interpret XML tags were needed in order to run .bpel files. This led others organizations to build their own engine to deploy their BPEL process.

With this plethoric number of BPEL engines now available, there is a need for us to propose a xBPEL engine architecture. This proposed architecture aims at facilitating the integration of the extension with existing BPEL engines. One point worthy to underline here is that, xBPEL is an extension of BPEL and not a new business language. So, it cannot work in standalone, but need to be coupled with the existing BPEL engines in order to provide full support for complex control flow patterns. With this in our mind, a proposed architecture for xBPEL will rely on existing BPEL engine. In fact, a layer, which establishes a relation between the two components – Generic extension mode for xBPEL – and – Generic specification model for BPEL, needs to be developed in order to facilitate the integration.

The two and very important components in the proposed xBPEL engine architecture are the "*generic extension model for xBPEL*" and the *"generic specification model for BPEL"*. These two components are herein represented in form of database and contain rules, constructs and attributes defined for each model. The generic BPEL component contains all the requirements for a standard .bpel file to execute, and the generic xBPEL component, contains the extension rules, constructs and attributes defined in this project. The two components are interrelated by a layer which aims at providing intercommunication support. At the upper layer of the two generic models, a user interaction layer provides a business architects with tools and utilities for building business processes more effectively. Figure 4.10 shows the proposed architecture.

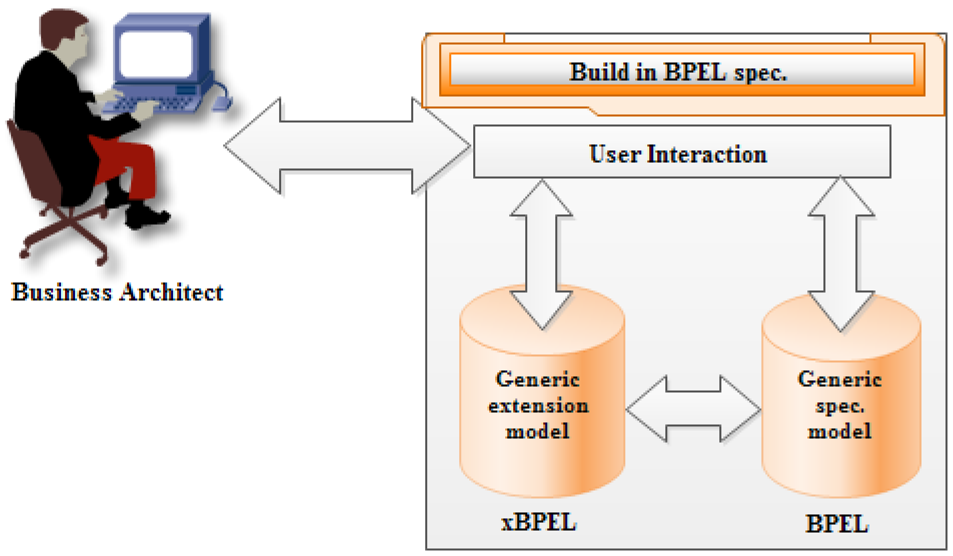


Figure 4.10: The Proposed BPEL/xBPEL engine integration

CHAPTER FIVE
SUMMARY, CONCLUSION AND RECOMMENDATIONS

5.0.Introduction

This chapter aims at giving summary of the achieved work, statement of the contribution to knowledge and suggestion for further works.

5.1.Summary

A wide majority of people are convinced today that application development and integration have evolved from the art of writing code to the art of assembling a set of services into new business capabilities. That is, application development and integration have been greatly elevated by the adoption of the architectural approach called Service-Oriented Architecture (SOA). SOA architectural approach conceptualizes functionality as services which refer to method of communications between two electronic devices over the World Wide Web. SOA strategy for using technologies to build business automation solution can be achieved through the use of web service composition such as Business Process Execution Language. This means in a nutshell that business process execution standards (BPEL) and Web Services (WS) have greatly reduced vendor lock-in and dramatically reduced costs by providing broader interoperability and reusability benefits. However, this adoption does not go without any problems. Emerged as a combination of two prior XML languages Web Service Flow Language (WSFL) and Microsoft XLANG, WS-BPEL is an XML based language which creates processes by using combination of the graph-oriented style of WSFL and the algebraic style of XLANG. This combination of XLANG and WSFL as two competing business languages has allowed BPEL to become the de facto industry standard for composing web services. However, in the beginning of this present decade, BPEL specification started losing its first admiration to the profit of Business Process Model and Notation (BPMN). In fact, a thorough review of the literature available on BPEL allowed this project to identify possible reasons that led to BPEL current downfall. One of these reasons that have been identified was BPEL's inability to support some workflow patterns.

In fact, business organizations are growing speedily and responding to changes in their business environment and their WS-BPEL business process will have to account for more and more complex message patterns. Current methods of orchestrating business process in WS-BPEL are limited: constructs, attributes and rules to support complex control flow patterns are not available. This inability and inadequacy to support complex control flow patterns make it substantially harder for WS-BPEL programmers or business architects to implement concise, yet highly-adaptable business processes.

Based on the aforementioned problem, we proposed in this work, solutions to these problems. The proposed extensions aim at providing support for complex patterns in terms of control flow patterns as identified in section 2.1.3. Reasons that explain why WS-BPEL 2.0 provides no support for some control flow patterns have been identified. Some of them rely in BPEL's inability to allow for two or more threads of execution to run through the same path in a single process instance; others rely on BPEL's inability to allow links to cross the boundaries of a loop; others reasons have been that BPEL, being a block structured base language, cannot capture unstructured cycles like arbitrary cycles. These and many other BPEL's inadequacies have been identified as problems causing the limitation of BPEL. Being the de facto business language for orchestration since 2004, BPEL language tends to disappear by giving hand to Business Process Model and Notation (BPMN) language which is a descriptive notation for modeling business processes that use standardized graphical symbols, meanings and logical connections while BPEL executes and describes processes using more technical tongue.

In this project, nine control flow patterns not supported by WS-BPEL 2.0 have been identified. This identification is based on the workflow patterns analysis carried by (W.M.P, Arthur, & Nick, Workflow Patterns Initiative, 2011) in 2011. In fact, two communication patterns and seven control flow patterns were identified in (W.M.P, Arthur, & Nick, Workflow Patterns Initiative, 2011) analysis. However, in this project we have concentrated on providing support for the control patterns. Furthermore, two out of the seven have been removed from the proposed extension because their implementation is based on the BPEL engine that executes the .bpel file. In a nutshell, five control flow patterns, namely, multi-merge, discriminator, arbitrary cycle, interleaved parallel routing and milestone patterns were studied. For each pattern, work-around solutions have been surveyed, studied and analyzed to see their feasibility. If the work-around solution is feasible, it has been adopted as part of the extension. This is the case with the milestone patterns in which two work-around solutions have been provided and partially or fully supported by current BPEL engines such as FLOWer and COSA (W.M.P, Arthur, & Nick, Workflow Patterns Initiative, 2011). For work-around solutions whose complexity increases with the number of activities, like the work-around solution proposed by (W.M.P, A.H.M, Kiepuszewski, & A.P., 2002), a new extension which reduces the complexity have been proposed. This is the case of the interleaved parallel routing. In addition to the extension thus proposed, a prototype implementation of each proposed extension was provided except for the milestone pattern where no direct support was provided but a work-around solution adopted.

At the end of this project, a xBPEL architecture for allowing the integration of xBPEL extension to existing BPEL engine was proposed. The design does not rely on any vendor-specific features but on WS-BPEL specification. In fact, two main components containing BPEL generic features and xBPEL features were defined. These two components interface with the business architect by using a user interaction component.

5.2.Contribution to knowledge

This study focused on adapting the BPEL language to support special control flow patterns. The major advantage of this approach is that there is no need to recreate a new set of languages or standards, but support is achieved at the implementation level of BPEL. All the associated developments and tools, such as BPEL engine and interactive development environment (IDE), can be reused for these patterns without any change. In this project, rules, constructs and attributes which need to be added into the BPEL specification were provided. This also allows architects to easily compose their business processes and make use of the extension in the situations whereby BPEL offers no support. Towards an easy and flexible business process composition is the main advantage of this project and there lies its contribution to knowledge. This project had intended to bring the BPEL specification back to the front burner in the business scene.

As it was noted in the introduction of this work, BPEL specification currently tends to vanish out of the web service scene because of many limitations. These limitations have led to BPEL's decline. However in this work, we have pondered upon a way to improve BPEL language in order to allow business managers to compose business processes more efficiently and therefore perpetuating BPEL as the de facto language for business composition. Also, the WS-BPEL language extension proposed in this work is minimally intrusive. In fact most of BPEL constructs such as *<flow>, <sequence>, <condition> etc.* were retained. As a consequence, WS-BPEL programmers do not have to learn all-new semantics for xBPEL. This approach also minimizes the charges needed in WS-BPEL engines to support xBPEL because only the proposed constructs, attributes rules will need to be implemented. Finally, since the only change this extension contributes is for unsupported control flow patterns and still rely completely on BPEL specification, programmers can continue to use their existing development skills in WS-BPEL and only make reference to xBPEL when need arises. This is a major advantage, since developers do not have to abandon their existing methodologies and development tools to use xBPEL.

5.3.Suggestions for Further Study

xBPEL is an extension of BPEL; not a full implementation of a BPEL specification. It cannot be used without being integrated with existing BPEL engines. It offers various extension points for complex control flow patterns. However much work is still needed in order to extend BPEL to all other Workflow Patterns not yet supported.

The main challenge for the future related to the BPEL specification is how to improve the scalability and interoperability across BPEL engines. Future work could explore how an instance process in execution can be transferred from one engine into the other without the need of redeploying the service.

REFERENCES

Akram, A., Meredith, D., & Allan, R. (2006). Evaluation of BPEL to Scientific Workflows. *Cluster Computing and the Grid, 2006. CCGRID 06* (pp. 269-274). Singapore: 6th IEEE International Symposium.

Allen, A. G. (2008). *Providing Context in WS-BPEL Processes.* Ontario, Canada: University of Waterloo.

Alonso, G., Machiraju, Casati, F., & Kanu, H. (2004). *Web Services: Concepts, Architectures and Applications.* Berlin: Springer.

Ambühler, T. (2005). *UML 2.0 Profile for WS-BPEL with Mapping to WS-BPEL.* Master Thesis, University of Stuttgart.

Amnuaykanjanasin, P., & Nupairoj, N. (2005). The BPEL orchestrating framework for secured grid services. *Information Technology: Coding and Computing, 2005* (pp. 348-353). Bangkok: IEEE International Conference, ITCC 2005.

C., P. (January 2003). *Web services orchestration: a review of emerging technologies, tools and standards.* Technical Paper.

Cristcost. (2007, December 11). *Comparison and Analogies between WSDL 1.1 and 2.0 structure.* Retrieved January 5, 2014, from Wikimedia Commons: http://commons.wikimedia.org

Decker, G., Kopp, O., Leymann, F., & Weske, M. (2007). BPEL4Chor: Extending BPEL for Modelling Choreographies. *Web Services, ICWS '07. IEEE International Conference*, (pp. 296-303). Salt Lake.

Deutch, D., & Milo, T. (2009). Evaluating TOP-K Queries over Business Processes. *IEEE 5th International Conference* (pp. 1195-1198). Shanghai: Data Engineering, 2009. ICDE '09.

Downey, P. (2007, January 13). *Whatfettle ~ BPEL Begone.* Retrieved March 30, 2014, from http://blog.whatfettle.com/2007/01/13/bpel-begone/

Dusan, S. B. (2013). *Extension of the iQ Workflow Engine.* Masaryk University - Faculty of Informatics.

Eclipse - The Object-XML component. (2014, March 11). *About Object-XML Mapping.* Retrieved March 18, 2014, from http://www.eclipse.org/eclipselink/documentation/2.5/concepts/blocks002.htm

Erik, C., Francisco, C., Greg, M., & Sanjiva, W. (2001, March 15). *Web Services Description Language (WSDL) 1.1.* Retrieved January 2014, 5, from W3C Recommendation: http://www.w3.org/TR/wsdl

Gregory, H., Christopher, G., & Gruia-Catalin, R. (2007). Extending BPEL for Interoperable Pervasive Computing. *Pervasive Services, IEEE International Conference* (pp. 204-213). Istanbul: IEEE Computer Society. doi:10.1109/PERSER.2007.4283918

Guangpu, Z., & Huilian, F. (Dec 2009). Publish and Subscribe Model Based on WS-Notification. *Eighth IEEE International Conference on* (pp. 857-858). Chengdu: Dependable, Autonomic and Secure Computing.

Hackmann, G., Haitjema, M., Gill, C., & Roman, G.-C. (2006). Sliver: A BPEL Workflow Process Execution Engine for Mobile Devices. *Springer*, 503-508.

Harish, G., & Markus, Z. (2006). *BPEL Cookbook.* Birmingham - Mumbai: PACKT Publishing.

Holmes, T., Vasko, M., & Dustdar, S. (2008). VieBOP: Extending BPEL Engines with BPEL4People. *Parallel, Distributed and Network-Based Processing, 16th Euromicro Conference* (pp. 547-555). Toulouse: IEEE Computer Society.

Huajian, Z., Xiaoliang, F., Ruisheng, Z., Jiazao, L., Zhili, Z., & Lian, L. (2008). Extending BPEL2.0 for Grid-Based Scientific Workflow Systems. *Asia-Pacific Services Computing Conference, APSCC '08. IEEE*, (pp. 757-762). Yilan.

Java Community. (n.d.). *JAX-WS.* Retrieved March 29, 2014, from https://jax-ws.java.net/nonav/2.2.6/docs/ch03.html

Jonathan, L., Ying-Yan, L., Shang-Pin, M., & Shin-Jie, L. (2009). BPEL Extensions to User-Interactive Service Delivery. *Journal of Information Science and Engineering 25*, 1427-1445.

Karastoyanova, D., & Leymann, F. (2009). BPEL'n'Aspects: Adapting Service Orchestration Logic. *Web Services, ICWS '09. IEEE International Conference* (pp. 222-229). Los Angeles: IEEE Computer Society.

Karastoyanova, D., Houspanossian, A., Cilia, M., & Leymann, F. (2005). Extending BPEL for run time adaptability. *EDOC Enterprise Computing Conference* (pp. 15-26). 9th IEEE International.

Kiepuszewski. (2003). *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows.* PhD Thesis, Queensland University of Technology.

Kiepuszewski, B., ter Hofstede, A., & Bussler, C. (2000). On structured workflow modelling. In I. B. Bergan (Ed.), *Proceedings of the 12th International Conference on Advanced Information Systems Engineering (CAiSE 2000). 1789.* Stockholm: Springer.

Kloppmann, M., D., K., F., L., G., P., & D., R. (2004). Business process choreography in websphere: Combining the power of BPEL and J2EE. *IBM Systems Journal, 43*, 270-296.

Ko, R. K., Lee, S. S., & E. W., L. (2009). Business Process Management (BPM) Standards: a survey. *Business Process Management Journal 15 (5),* 744-791.

Koegel, M. (2013, July 16). *EMF Tutorial.* Retrieved February 20, 2014, from http://eclipsesource.com

König, D., Lohmann, N., Moser, S., Stahl, C., & Wolf, K. (2008). Extending the compatibility notion for abstract WS-BPEL processes. *Proceedings of the 17the International Conference on World Wide Web*, (pp. 785-794). Beijing.

Marwan, S., Stu, J., Dock, A., Paul, S., & Paul, D. (2001, April 11-12). World Wide Web Consortium - Workshop on Web services. San Jose, CA - USA.

Mendling, J., Strembeck, M., Stermsek, G., & Neumann, G. (2004). An Approach to Extract RBAC Models from BPEL4WS Processes. *in Proceedings of the 13th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WET ICE).* Modena.

Michael, H. (2005). *Essential Business Process Modeling.* O'Reilly.

Michael, R. (2009). *Extending BPEL with transitions that can loop.* Active Endpoints Inc.

Michiel, K., Chang-ai, S., Marco, S., & Paris, A. (2009). VxBPEL: Supporting variability for Web services in BPEL. *Information and Software Technology, ELSEVIER*, 258-269.

Newcommer, E., & Lomow, G. (2004). *Understanding Service-Oriented Architecture (SOA) with Web Services.* Addison-Wesley Professional.

OASIS - BPEL Technical Committee. (2007, April 11). *Web Services Business Process Execution Language Version 2.0.* Retrieved January 14, 2014, from OASIS Standard -: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf

OASIS. (2004, October 19). *UDDI Specification TC.* Retrieved December 30, 2013, from UDDI Spec Technical Committee Draft: http://uddi.org/pubs/uddi_v3.htm

OASIS SOA Reference Model (RM). (2012, December 4). *Reference Architecture Foundation for Service Oriented Architecture Version 1.0.* Retrieved December 31, 2013, from SOA Committee Specification 01: docs.oasis-open.org/soa-rm/soa-ra/v1.0/cs01/soa-ra-v1.0-cs01.pdf

OASIS UDDI Specification. (2004). *Introduction to UDDI: Important Features and Functional Concepts*. Retrieved from OASIS: www.oasis-open.org

Onyeka, E., & Sadjadi, M. (2005). *Composing Aggregate Web Services in BPEL.* Florida: Autonomic Computing Research Laboratory, Florida International University.

Oracle - JAXB Architecture. (2010). *The Java EE 5 Tutorial.* Retrieved March 18, 2014, from Architecture Overview: http://docs.oracle.com/javaee/5/tutorial/doc/bnazg.html

Oracle. (2013). *Using JAXB Data Binding.* Retrieved March 18, 2014, from http://docs.oracle.com/cd/E24329_01/web.1211/e24964/data_types.htm

Petia, W., Wil M.P., v., & Marlon, D. (2002). *Pattern Based Analysis of BPEL4WS.* Queensland University of Technology.

Philip, M. (2006). *Design and Implementation of a Framework for testing BPEL compositions.* Leibniz University Hannover.

Roberto, C., Jean-Jacques, M., Arthur, R., & Sanjiva, W. (2007, June 26). *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language.* Retrieved January 5, 2014, from W3C Recommendation: http://www.w3.org/TR/wsdl20/

Shin, N. (2006). Model-Checking Behavioral Specification of BPEL Applications. *Proceedings of the International Workshop on Web Languages and Formal Methods* (pp. 89-105). Tokyo: Science Direct.

Slominski, A. (2007). Adapting BPEL to Scientific Workflows. *Springer*, 208-226.

SOAP Specifications - World Wide Web Consortium. (2007, April 27). *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition).* Retrieved December 30, 2013, from W3C Recommendation: http://www.w3.org/TR/soap12-part1/

Tim, B.-L. (2009, August 27). *Web Services.* Retrieved December 30, 2013, from World Wide Web Consortium - W3C: http://www.w3.org/DesignIssues/

Tim, D., Matthew, S., & Bernd, F. (2008). Composition and Execution of Secure Workflows in WSRF-Grids. *8th IEEE International Symposium* (pp. 122-129). Lyon: Cluster Computing and the Grid, 2008. CCGRID '08.

Tim, D., Thomas, F., Sergej, H., Ernst, J., & Bernd, F. (2007). Grid Workflow Modelling Using Grid-Specific BPEL Extensions. *Proceedings of German e-Science Conference (GES).* Marburg: GES.

Tony, A., Francisco, C., Hitesh, D., Yaron, G., Johannes, K., Frank, L., . . . Sanjiva, W. (2003, May 5). *Business Process Execution Language for Web Services, Version 1.1.* Retrieved December 15, 2013, from IBM Software: ftp://www6.software.ibm.com/software/developer/library/ws-bpel11.pdf

Turner, K. J. (2005). Formalising web services. In F. Wang (Ed.), *Proceedings of Formal Techniques for Networked and Distributed Systems (FORTE XVIII)* (pp. 473–488). Berlin: Springer.

UDDI Consortium. (2001, November 14). *UDDI Executive White Paper.* Retrieved December 30, 2013, from UDDI Executive White Paper: uddi.org/pubs/UDDI_Executive_White_Paper.pdf

Vasko, M., & Dustdar, S. (2004). An Analysis of Web Services Workflow Patterns in Collaxa. *Springer,* 1-14.

W.M.P, v., A.H.M, t., Kiepuszewski, B., & A.P., B. (2002). *Workflow Patterns: Technical Report.* Faculty of IT, Queensland University Technology.

W.M.P, v., Arthur, H., & Nick, R. (2011). *Workflow Patterns Initiative*. Retrieved January 4, 2014, from http://www.workflowpatterns.com/evaluations/standard/

W.M.P., v., Arthur, H., & Nick, R. (2011). *Workflow Patterns*. Retrieved from www.workflowpatterns.com

W3C Web Services Addressing - Core. (2006, May 9). *Web Services Addressing 1.0 - Core.* Retrieved January 7, 2014, from World Wide Web Consortium: http://www.w3.org/TR/ws-addr-core/

W3C Web Services Addressing - Metadata. (2007, September 4). *Web Services Addressing 1.0 - Metadata.* Retrieved January 9, 2014, from W3C Recommendation: http://www.w3.org/TR/ws-addr-metadata/

W3C Working Group. (2004a, February 11). *Web Services Architecture Requirements.* Retrieved December 30, 2013, from W3C: http://www.w3.org/TR/wsa-reqs/

W3C Working Group. (2004b, February 11). *Web Services Glossary.* Retrieved January 7, 2014, from W3C: http://www.w3.org/TR/ws-gloss/

Weerawarana, S., Curbera, F., Leymann, F., Tony, S., & Ferguson, D. F. (2005). *Web Services Platform Architecture.* Prentice Hall PTR.

World Wide Web Consortium - SOAP. (2007, April 27). *SOAP Specifications - Version 1.2 Part 1: Messaging Framework (Second Edition).* Retrieved December 30, 2013, from W3C Recommendation: http://www.w3.org/TR/soap12-part1/

World Wide Web Consortium - SOAP. (2007, April 27). *W3C Recommendation.* Retrieved December 30, 2013, from SOAP Specification Version 1.2 Part 2: Adjuncts (Second Edition): http://www.w3.org/TR/soap12-part2/

World Wide Web Consortium. (2004, February 11). *Web Service Architecture.* Retrieved December 30, 2013, from W3C: http://www.w3.org/TR/ws-arch/

World Wide Web Consortium. (2007, April 27). *SOAP Specifications - Version 1.2 Part 1: Messaging Framework (Second Edition).* Retrieved December 30, 2013, from W3C Recommendation: http://www.w3.org/TR/soap12-part1/

World Wide Web Consortium. (2007, April 27). *W3C Recommendation.* Retrieved December 30, 2013, from SOAP Specification Version 1.2 Part 2: Adjuncts (Second Edition): http://www.w3.org/TR/soap12-part2/

Xiang, F., Tevfik, B., & Jianwen, S. (2004). Analysis of interacting BPEL web services. *Proceedings of the 13th International Conference on World Wide Web*, (pp. 621-630). New York.

Yi, Q., Yuming, X., Zheng, W., Geguang, P., Huibiao, Z., & Chao, C. (2007). Tool Support for BPEL Verification in ActiveBPEL Engine. *Software Engineering Conference,2007. ASWEC 2007. 18th Australian* (pp. 90-100). Melbourne, Vic.: IEEE Conference Publications.

Yi, Q., Yuming, X., Zheng, W., Geguang, P., Huibiao, Z., & Chao, C. (2007). Tool Support for BPEL Verification in ActiveBPEL Engine. *Software Engineering Conference. ASWEC 2007, 18th Australian* (pp. 90-100). Melbourne: IEEE Computer Society.

Yuli, V. (2007). *SOA and WS-BPEL, Composing Service-Oriented Solutions with PHP and ActiveBPEL.* Birmingham-Mumbai: PACKT Publishing.

APPENDIX A

```
 Author: Jean-Paul Ainam,
Date: 15 March 2014
******************************
```

```
public class Webservicex implements Runnable{
          private String result = null;
          public boolean hasAnswered = false;
          private GeoIPServiceSoap geo = null;

          public Webservicex(){
                    GeoIPService geoservice = new GeoIPService();
                    geo =  geoservice.getGeoIPServiceSoap();
          }
          public String getResult(){
                    return result;
          }
          @Override
          public void run() {
                while(!hasAnswered){
                    GeoIP ip = geo.getGeoIP("214.12.211.18");
                    result = ip.getCountryName();
                    if(!result.isEmpty())
                    hasAnswered = true;
                    else
                    continue;
                }

          }
```

Cdyne.java class

```
public class Cdyne implements Runnable{
       private String result = null;
       public boolean hasAnswered = false;
       private IP2GeoSoap geosoap = null;

       public Cdyne(){
              IP2Geo geoservice = new IP2Geo();
              geosoap = geoservice.getIP2GeoSoap();
       }
       public String getResult(){
                          return result;
       }
       @Override
       public void run() {
              while(!hasAnswered){
              IPInformation infosip = geosoap.resolveIP("214.12.211.18", "0");
                 result = infosip.getCountry();
                 if(!result.isEmpty())
                 hasAnswered = true;
                 else
                 continue;
              }

         }
}
```

TestGeoIP.java class

```java
package geo.ws.testing;

public class TestGeoIP implements Runnable{
     /**
          * Thread to run xService provider
     */
    private Thread xThread = null;
    /**
     * Thread used to run cdyne IP service provider
     */
    private Thread gThread = null;
    /**
     * determine which webservice has answered first
     * 1 = WebServicex GeoIP Service
     * 2 = Cdyne GeoIP Service
     */
    public int who;

    public Webservicex servicex = null;

    public Cdyne cdyne = null;
    private String result;

    private long startTime;
    private long endTime;

    public TestGeoIP(){
        servicex = new Webservicex();
        cdyne = new Cdyne();

        xThread = new Thread(servicex);
        gThread = new Thread(cdyne);

        startTime = System.currentTimeMillis();
        xThread.start();
        gThread.start();
    }
    /**
     * ends {@link #xThread} and {@link #gThread}
     */
    public void stop(){
        while(true){
            try {
                xThread.join();
                gThread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            break;
        }
        xThread = null;
        gThread = null;
    }
    @Override
    public void run() {
        Thread thisThread = Thread.currentThread();
        while(true){
            System.out.println("...");
            if(servicex.hasAnswered){
                System.out.println("Enter " + servicex.getResult());
                who = 1;
                result = servicex.getResult();
                endTime = System.currentTimeMillis();
                stop();
                break;
            }else{
                if(cdyne.hasAnswered){
                    who = 2;
                    result = cdyne.getResult();
                    endTime = System.currentTimeMillis();
                    stop();
                    break;
                }
            }
         continue;
       }
     }
    public static void main(String[] args) {
        TestGeoIP test = new TestGeoIP();
        test.run();
        String str = null;
        /* Convert the time to seconds */
        long last = (test.endTime - test.startTime);
        System.out.println("last " + last);
        if(test.who == 1){
            str = "Web Servicex Geo IP : " + test.result + "\t Time = " + last;
       }else
            str = "Cdyne Geo IP : " + test.result + "\t Time = " + last;
       JOptionPane.showMessageDialog(null, str);
    }
}
```

Ipr.wsdl

```
<?xml version="1.0" encoding="UTF-8"?>
        <definitions xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
        wssecurity-utility-1.0.xsd" xmlns:wsp="http://www.w3.org/ns/ws-policy"
        xmlns:wsp1_2="http://schemas.xmlsoap.org/ws/2004/09/policy"
        xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="http://db.ipr.com/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns="http://schemas.xmlsoap.org/wsdl/" targetNamespace="http://db.ipr.com/"
        name="IPRConnectImplService">
<types>
    <xsd:schema>
        <xsd:import namespace="http://db.ipr.com/"
                  schemaLocation="http://localhost:9091/ws/ipr?xsd=1"/>
    </xsd:schema>
</types>
<message name="updateAccount">
     <part name="parameters" element="tns:updateAccount"/>
</message>
<message name="updateAccountResponse">
     <part name="parameters" element="tns:updateAccountResponse"/>
</message>
<portType name="IPRConnect">
    <operation name="updateAccount">
          <input wsam:Action=http://db.ipr.com/IPRConnect/updateAccountRequest
          message="tns:updateAccount"/>
          <output wsam:Action=http://db.ipr.com/IPRConnect/updateAccountResponse
          message="tns:updateAccountResponse"/>
    </operation>
</portType>
<binding name="IPRConnectImplPortBinding" type="tns:IPRConnect">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="updateAccount">
        <soap:operation soapAction=""/>
        <input>
             <soap:body use="literal"/>
        </input>
        <output>
             <soap:body use="literal"/>
        </output>
    </operation>
</binding>
<service name="IPRConnectImplService">
   <port name="IPRConnectImplPort" binding="tns:IPRConnectImplPortBinding">
         <soap:address location="http://localhost:9091/ws/ipr"/>
   </port>
</service>
</definitions>
```

GeoIP.java

```java
package net.webservicex;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlType;

/**
 * <p>Java class for GeoIP complex type.
 * 
 * <p>The following schema fragment specifies the expected content contained within this 
class.
 * 
 * <pre>
 * &lt;complexType name="GeoIP">
 *   &lt;complexContent>
 *     &lt;restriction base="xsd:anyType">
 *       &lt;sequence>
 *         &lt;element name="ReturnCode" type="xsd:int"/>
 *         &lt;element name="IP" type="xsd:string" minOccurs="0"/>
 *         &lt;element name="ReturnCodeDetails" type="xsd:string" minOccurs="0"/>
 *         &lt;element name="CountryName" type="xsd:string" minOccurs="0"/>
 *         &lt;element name="CountryCode" type="xsd:string" minOccurs="0"/>
 *       &lt;/sequence>
 *     &lt;/restriction>
 *   &lt;/complexContent>
 * &lt;/complexType>
 * </pre>
 * 
 */
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "GeoIP", propOrder = {
    "returnCode",
    "ip",
    "returnCodeDetails",
    "countryName",
    "countryCode"
})
public class GeoIP {

    @XmlElement(name = "ReturnCode")
    protected int returnCode;
    @XmlElement(name = "IP")
    protected String ip;
    @XmlElement(name = "ReturnCodeDetails")
    protected String returnCodeDetails;
    @XmlElement(name = "CountryName")
    protected String countryName;
    @XmlElement(name = "CountryCode")
    protected String countryCode;

}
```

IPInformation.java

```java
package com.cdyne.ws;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlType;

/**
 * <p>Java class for IPInformation complex type.
 * 
 * <p>The following schema fragment specifies the expected content contained within this 
class.
 * 
 * <pre>
 * &lt;complexType name="IPInformation">
 *   &lt;complexContent>
 *     &lt;restriction base="xsd:anyType">
 *       &lt;sequence>
 *         &lt;element name="City" type="xsd:string" minOccurs="0"/>
 *         &lt;element name="StateProvince" type="xsd:string" minOccurs="0"/>
 *         &lt;element name="Country" type="xsd:string" minOccurs="0"/>
 *         &lt;element name="Organization" type="xsd:string" minOccurs="0"/>
 *         &lt;element name="Latitude" type="xsd:double"/>
 *         &lt;element name="Longitude" type="xsd:double"/>
 *         &lt;element name="AreaCode" type="xsd:string" minOccurs="0"/>
 *         &lt;element name="TimeZone" type="xsd:string" minOccurs="0"/>
 *         &lt;element name="HasDaylightSavings" type="xsd:boolean"/>
 *         &lt;element name="Certainty" type="xsd:short"/>
 *         &lt;element name="RegionName" type="xsd:string" minOccurs="0"/>
 *         &lt;element name="CountryCode" type="xsd:string" minOccurs="0"/>
 *       &lt;/sequence>
 *     &lt;/restriction>
 *   &lt;/complexContent>
 * &lt;/complexType>
 * </pre>
 *
 */
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "IPInformation", propOrder = {
    "city",
    "stateProvince",
    "country",
    "organization",
    "latitude",
    "longitude",
    "areaCode",
    "timeZone",
    "hasDaylightSavings",
    "certainty",
    "regionName",
    "countryCode"
})
public class IPInformation {
    @XmlElement(name = "City")
    protected String city;
    @XmlElement(name = "StateProvince")
    protected String stateProvince;
    @XmlElement(name = "Country")
    protected String country;
    @XmlElement(name = "Organization")
    protected String organization;
    @XmlElement(name = "Latitude")
    protected double latitude;
    @XmlElement(name = "Longitude")
    protected double longitude;
    @XmlElement(name = "AreaCode")
    protected String areaCode;
    @XmlElement(name = "TimeZone")
    protected String timeZone;
    @XmlElement(name = "HasDaylightSavings")
    protected boolean hasDaylightSavings;
    @XmlElement(name = "Certainty")
    protected short certainty;
    @XmlElement(name = "RegionName")
    protected String regionName;
    @XmlElement(name = "CountryCode")
    protected String countryCode;
}
```

ArrayOfString.java

```java
package com.microsoft.schemas._2003._10.serialization.arrays;
import java.util.ArrayList;
import java.util.List;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlType;

/**
 * <p>Java class for ArrayOfstring complex type.
 * <p>The following schema fragment specifies the expected content contained within this
class.
 * <pre>
 * &lt;complexType name="ArrayOfstring">
 *   &lt;complexContent>
 *     &lt;restriction base="xsd:anyType">
 *       &lt;sequence>
 *         &lt;element name="string" type="xsd:string" maxOccurs="unbounded"
minOccurs="0"/>
 *       &lt;/sequence>
 *     &lt;/restriction>
 *   &lt;/complexContent>
 * &lt;/complexType>
 * </pre>
 *
 */
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "ArrayOfstring", propOrder = {
    "string"
})
public class ArrayOfstring {
    @XmlElement(nillable = true)
    protected List<String> string;

    @Override
    public String toString() {
          String str = "";
          for(String s : this.string){
          str += s + "\n";
          }
          return str;
    }

/**
     * Gets the value of the string property.
     *
     */
    public List<String> getString() {
        if (string == null) {
            string = new ArrayList<String>();
        }
        return this.string;
    }
}
```