Gottfried Wilhelm
Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Software Engineering

Design and Implementation of a Framework for Testing BPEL Compositions

Masterarbeit

im Studiengang Informatik

von

Philip Mayer

Prüfer: Prof. Dr. Kurt Schneider
Zweitprüfer: Prof. Dr. Rainer Parchmann
Betreuer: Dipl.-Wirt.-Inform. Daniel Lübke

Hannover, 11. September 2006

Abstract

The Business Process Execution Language (BPEL) is emerging as the new standard in Web service composition. As more and more workflows are modeled using BPEL, unit-testing these compositions becomes increasingly important. However, little research has been done in this area and no frameworks comparable to the xUnit family are available. In this thesis, a layer-based approach to creating frameworks for repeatable, white-box BPEL unit testing is presented. Based on this approach, the concrete BPEL composition testing framework *BPELUnit* is developed. This framework uses a specialized BPEL-level testing language and literal XML data to describe interactions with a BPEL process to be carried out in a test case, and supports automated BPEL deployment/undeployment and test execution.

Contents

1 Introduction

1.1 Motivation

One of the more important areas of enterprise software development is the realization of business processes in software. However, many companies have employed different technologies for building enterprise software through the years, leading to heterogeneous software environments which complicate system integration, and thus hinder the development of flexible and maintainable business processes.

With the advent of the Service-Oriented Architecture (SOA), formerly proprietary software systems are being opened and made available as services in a homogeneous way. According to Gartner (Natis 2003), SOAs are one of the most promising architectural styles for the development of enterprise software in the next years. The most dominant implementation of SOAs is the Web service standards family. Web services can be characterized as software components accessible via the Internet, which encapsulate a certain task and may be integrated into more complex, distributed applications (Alonso et al. 2004, Weerawarana et al. 2005).

Composition of such Web services is seen as one way for achieving the goal of flexible and maintainable business processes. The Business Process Execution Language (BPEL) (Andrews et al. 2003) has been created specifically for this purpose: BPEL compositions, described in XML, form an executable program which interacts with other Web services, combining them to accomplish a certain task. The composition is recursive, as BPEL compositions are themselves exposed as Web services.

BPEL is backed by major industry players and is seen by many as the emerging dominant standard for Web service composition. As more and more compositions are modeled using BPEL, ensuring good-quality BPEL code becomes critical. One key approach to ensuring software quality is software testing in its many forms, among them repeatable, white-box unit testing as a code-centric approach.

In other areas, the need for automated and repeatable testing is already widely recognized, for example in the Extreme Programming community (Beck 2000) and in the area of Test-Driven Development (Beck 2003). Unit testing has already been proven to improve quality in practice (Ellims et al. 2004). Consequently, there are many unit testing frameworks available for all kinds of programming languages (Hamill 2004).

However, little research has been done in the area of BPEL composition testing. Existing research papers focus on the theoretical side of testing (Li et al. 2005), and BPEL editors currently available – like the Oracle BPEL process manager, the ActiveBPEL Designer or the preview version of Sun's NetBeans 5.5 – offer only manual black box testing.

The intent of this thesis is to extend the reach of software testing, and in particular, automated, repeatable white-box unit testing, to BPEL compositions. This leads to the main goal of this thesis: *The creation of a framework for testing BPEL compositions*.

1.2 Problem description

The creation of such a framework involves three steps: Firstly, an exploration of the realm of BPEL composition testing to lay the foundations for a testing framework; secondly, a discussion of possible approaches to the development of such a framework, and finally the design and implementation of a concrete framework, enabling users to actually run tests and verify the approach.

In the first step, some decisions are made with regard to the underlying principles of BPEL unit testing. In particular, the following three definitions are discussed:

- The definition of the *test approach*, which includes requirements of BPEL unit testing, and necessary qualifications of testers.
- The definition of a *unit*, i.e. the component under test in a BPEL unit test.
- The definition of a *unit test case*, i.e. the general approach to interacting with the unit under test.

The next step consists of a discussion of possible approaches to the creation of BPEL unit testing frameworks. In particular, a general *architecture* for such frameworks is discussed, which captures the inherent problems of BPEL unit testing, and serves as a foundation for the development of concrete software systems to be used to create and execute unit tests as well as gather and present the results.

When considering such an architecture, the requirements identified in the first step must be taken into account – in particular, the special role of the BPEL programming language. Contrary to conventional multi-purpose programming languages, BPEL is a highly domain-specific language and resides on top of a complex stack of other technologies (the Web service stack). This leads to three important challenges for BPEL composition testing:

- BPEL programs run inside a *BPEL middleware*, which provides the necessary runtime environment for a BPEL program, like the ability to send and receive calls to or from other Web services. A BPEL unit test must be able to interface with the BPEL middleware in order to test a BPEL composition.
- Being Web services, BPEL compositions deal with *XML data* as their native data format. A BPEL unit test must provide ways of specifying such XML data and comparing BPEL data with test data.
- BPEL compositions are intended to integrate distributed software components, often with the use of asynchronous calls, creating an *inherent parallel design* in many BPEL processes. Although the use of parallel threads is also possible in conventional programming languages, special tools must be used to test modules which make extensive use of such threads. A BPEL composition testing framework needs to include such functionality to start with.

Finally, with a general architecture of BPEL unit testing frameworks in place, the focus can move to the creation of a concrete testing framework and the more practical requirements for such a framework. Programming in BPEL is a complex task, and requires the creation of many artifacts, possibly with different tools. In order to be successful, a unit test framework must be simple to use, not require a tedious "mode switch" between programming and testing, and ideally employ existing, well-known designations for testing artifacts.

Simplicity of the testing framework and naming compatibility to existing approaches are therefore key requirements for the testing framework presented in later chapters, along with the commitment to all of the special requirements for BPEL composition testing mentioned above.

In this thesis, the concrete BPEL unit testing framework *BPELUnit* is developed, which is intended as an extension of the xUnit family into the area of Web service composition testing, enabling developers to take the testing approach familiar from the xUnit family frameworks with them when moving to BPEL.

1.3 Structure

This thesis is structured as follows:

In chapter 2, the theoretical underpinnings of this thesis are presented, starting with a definition of a SOA and a description of the Web service stack with particular emphasis on the three standards SOAP, WSDL, and WS-Addressing. An in-depth review of the BPEL programming language follows, and the chapter is concluded with a description of software testing, and of unit testing in particular.

Chapter 3 then presents BPEL composition testing approaches, including a general testing framework architecture. This chapter also introduces the concrete framework BPELUnit. In chapter 4, the technical design and implementation of BPELUnit and its supporting tools are presented along with a thorough description of all extension points to enable the development of additional plug-ins and tools.

Chapter 5 presents two example BPEL processes and their test suites. Finally, chapter 6 concludes the thesis with a summary and an outlook on future work.

2 Underlying concepts

This chapter introduces the underlying concepts and technologies which form the basis for the BPEL composition testing framework presented in later chapters. In section 2.1, the Service-Oriented Architecture (SOA) is presented, followed by Web services in section 2.2. The Business Process Execution Language (BPEL) is detailed in section 2.3. Last but not least, section 2.4 discusses software testing.

2.1 The 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. Although it is hard to find an exact definition, 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.

SOAs are closely tied to the hope of being able to re-structure the intra- and inter-enterprise software landscape to allow greater flexibility, thus being able to respond more quickly to changing business requirements.

Distributed architectures and integration methods have been around for some time, and SOA has evolved out of these methods rather than being a completely new concept. In fact, SOAs take intra-enterprise integration systems to a new level – as companies are outsourcing parts of their business or are cooperating with partners, their information systems also grow across company borders. Additionally, SOAs enable re-use of existing applications by wrapping them as services.

As SOAs also features service integration, one key application area is the realization of *business processes* as a composition of services, thereby placing SOAs at the very heart of enterprise IT. For example, an analysis by Gartner (Natis 2003) lists the following benefits of SOAs for enterprises:

- Incremental development and deployment of business software
- Reuse of business components in multiple business experiences
- Low-cost assembly of some new business processes
- Clarity of application topology

Gartner also makes the following prediction for the role of SOAs in the near future:

By 2008, SOA will be a prevailing software-engineering practice, ending the 40-year domination of monolithic software architecture (0.7 probability).

With this background, it is easy to see why SOAs have attracted so much attention. Nevertheless, a concrete, standards-based implementation must be agreed upon to benefit from the promises of this architecture. The Web service technology is such a standard, which is detailed in the next section.

2.2 Web services

Web services are, arguable, the most important realization of a SOA today. The Web service architecture consists of over a dozen standards, which are managed by standardization bodies like W3C (W3C 2006b) and OASIS (OASIS 2006). The Web service movement is backed by large industry players like IBM, Microsoft, Sun, and others.

Although Web services are just one possible realization of a SOA, the terms Web services and SOA are often (incorrectly) used as synonyms. This underlines the impact the Web services movement has had on the market.

There are many different definitions of a Web service. Even the W3C, which is involved in many of the basic Web service standards, has two definitions available (W3C 2004b, W3C 2004a). The first definition 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 2004b).

A few things should be noted with regard to this definition. Firstly, the definition describes a Web service as a software *application*, which can basically mean anything from a small script to a large, server-spanning enterprise software system. This is intentional – the Web services technology does not presume any specific size of Web services; this lies in the responsibility of the interface designer. A Web service encapsulates a task, which can be of any size.

Secondly, a Web service is self-describing, and must be discoverable, with the use of XML. The use of XML is important as it is a text-only, open standard, and if properly designed, XML documents may be read by humans and easily manipulated with generic XML software.

Thirdly, a Web service does not interact with humans, but with other software agents. This fact has led to much confusion in the beginning of the Web services movement, as people tended to think of Web *pages* when reading about Web services.

Finally, as can be seen from this definition and in fact from the name Web services itself, the Web plays a major role in the Web service standards. The definition not only mentions the concept of a URI, but also *Internet-based protocols* as the message exchange layer.

W3C has another definition available, which already includes the names of some of the most important W3C Web service standards, which are described below:

A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards (W3C 2004a).

Web services offer a significant step forward from existing middleware and EAI solutions due to three aspects, which are identified in Alonso et al. 2004:

- **Service Orientation**. As a realization of a SOA, service orientation comes natural to Web services. However, inherent loose coupling between the components of a system – each service is independently implemented – is a major change from older middleware systems.
- **Peer-to-Peer Middleware Protocols**. The runtime environment of Web services – which consists of inter-enterprise space – offers no place for a central coordinator for management of resources and locks. Instead, the various peers must be able to agree upon such things on a bilateral basis.
- **Standardization**. The aim of Web services is to offer cross-enterprise integration, which necessitates industry-wide standards. The Web service movement has been successful in many ways in introducing such standards by employing standardization bodies and major industry players.

The Web service movement began with the publication of the three initial Web service standards in mid to late 2000, which still form the basic triangle of the Web service architecture:

- **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.

- **UDDI**, the Universal Discovery, Description, and Integration Service for Web services, handles dynamic discovery of Web services.

The following diagram shows the Web service architecture stack. Note how SOAP and WSDL are part of the horizontal layering, while UDDI provides a vertical service throughout the stack.

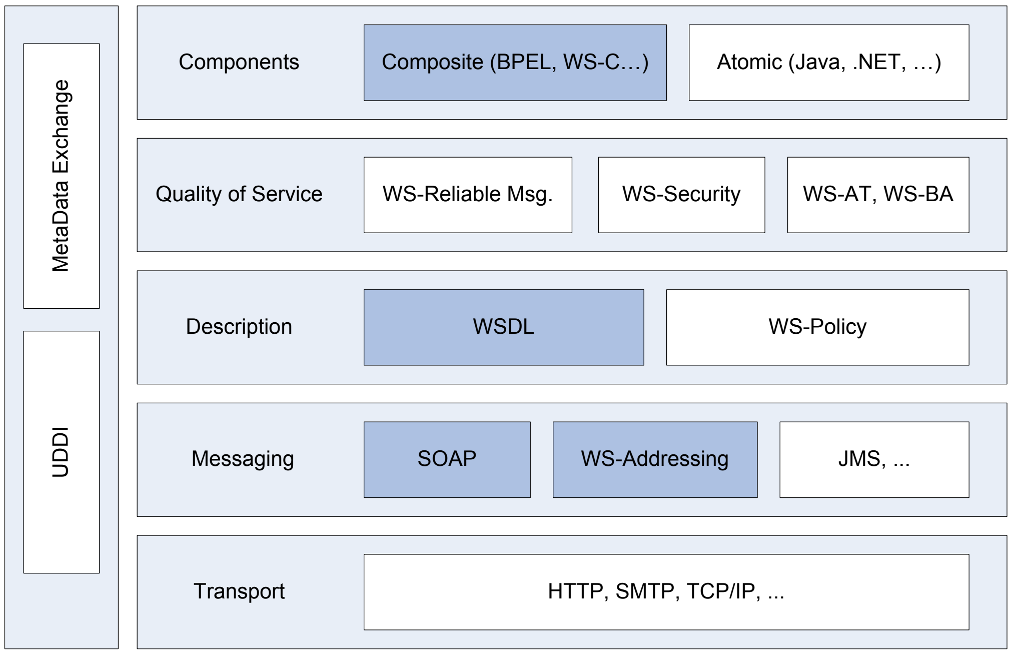


Figure 1: The Web Service Architecture Stack

This thesis 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 (Andrews et al. 2003); additionally, the WS-Addressing specification is required to enable asynchronous messaging. Therefore, these three specifications will be explained in detail in the next sections. W3C 2004a offers more information on other Web service standards.

The Web services technology is, as pointed out above, fairly new, with its beginnings in the year 2000. Although Web services are spreading quickly and are seen by many as some sort of silver bullet, even the basic standards and also the actual implementations still have severe interoperability problems. In section 2.2.4, some light is shed on these problems and possible remedies.

2.2.1 SOAP

SOAP (W3C 2003) is a protocol for exchanging messages between peers in a distributed environment. SOAP is based on XML and defines a standardized message format, a processing model, a mechanism for binding messages to network protocols for transport, and a set of conventions, defining how to map application data into messages.

SOAP was initially created in 1999 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).

The basic capabilities of SOAP include the following:

- **Standardized Message Format**. SOAP specifies how information is packaged into a standardized XML document (a SOAP Message) to be transferred in a communication.

- **Mapping Conventions**. The SOAP specification contains a set of conventions for mapping application data into the SOAP messages, for example for specifying a remote procedure call.

- **Processing Model**. The processing model defines roles for senders and receivers of SOAP messages, specifying which parts of a message must be processed by a role.

- **Network Protocol Bindings**. SOAP specifies bindings, which describe how SOAP messages are to be transported over HTTP, SMTP, and other transport protocols.

The SOAP communication protocol is one-way, stateless, and exchanges information using *messages*. A SOAP message is, in essence, just a standardized XML document, containing some metadata, and incorporating the actual application data. Despite the set of conventions for mapping the application data for SOAP transfer, SOAP does not attempt to interpret this data or perform any other semantic operations; the data is simply payload of the SOAP message.

2.2.1.1 SOAP message processing

As can be seen in Figure 2, 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.

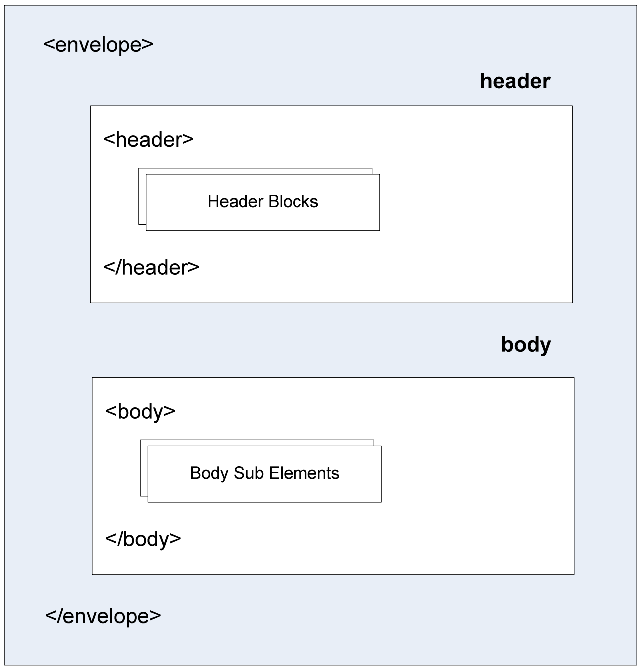


Figure 2: Structure of a SOAP Message

A SOAP header consists of a number of *header blocks*, which are the first-level children of the header. Likewise, the SOAP body consists of a number of *body sub-elements*.

SOAP messages are transmitted between SOAP nodes. A SOAP node can send and/or receive messages. There is one *initial sender* of a SOAP message and an *ultimate receiver*, and any number (including zero) of *intermediaries*, which both receive and send messages.

The main information which the message carries – the application data – is sent by the initial sender and targeted at the ultimate receiver; this information is placed in the SOAP body. Any other information which might be needed on the way – for example, for routing the message – is placed in the SOAP header. The header may, in principle, be read and written by any node processing the message.

2.2.1.2 The SOAP header

The *SOAP processing model* mainly contains instructions about the SOAP header; in particular, how SOAP nodes must handle the header blocks. The blocks are XML fragments in their own namespaces and are in no way defined by SOAP; rather, they use an application- or environment-specific format. However, the root node of a header block may contain attributes from the SOAP namespace, indicating the intended use of this header block.

Three attributes are defined in the SOAP specification:

- **role**. The role attribute identifies the role played by the intended target of the header block. Any SOAP node may choose to play a role; how a SOAP node takes on a role is not specified by SOAP. The role attribute may contain arbitrary URLs; however, three roles are described within the SOAP specification:

- **none**: If the block is assigned to the *none* role, the block should not be read by any node (the data may be required for processing of other blocks, though).
- **ultimateReceiver**: If the block is assigned to the *ultimateReceiver* role, the block is only to be read by the last receiver, not any intermediary node.
- **next**: If the block is assigned to the *next* role, it is intended to read by any intermediary node, as every node is a "next node" during processing.

- **mustUnderstand**. If the *mustUnderstand* attribute is set to true, a target node failing to understand the contents of the header must stop processing and generate a fault.
- **relay**. Normally, a SOAP processing node removes a header block once it is processed. However, by specifying the *relay* attribute, a node can choose not to process a header block, but rather send it on to the next node.

One of the many uses of SOAP headers is to include callback information for asynchronous calls. The WS-Addressing specification makes use of this mechanism, see section 2.2.3.

Figure 3 shows an example of a SOAP message (W3C 2003). It contains a header with two header blocks, whose content is omitted for brevity, and the mandatory SOAP body, whose content is also omitted. The first header block carries a custom role (not defined by SOAP), which must be understood by the node it applies to; the second block carries the next role, but may be relayed.

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
     <env:Header>
          <p:oneBlock xmlns:p="http://example.com/p"
               env:role="http://example.com/Log"
               env:mustUnderstand="true">
               ...
          </p:oneBlock>
          <q:anotherBlock xmlns:q="http://example.com/q"
               env:role="http://www.w3.org/.../role/next"
               env:relay="true">
               ...
          </q:anotherBlock>
     </env:Header>
     <env:Body>
          ...
     </env:Body>
</env:Envelope>
```

Figure 3: A SOAP Message

2.2.1.3 The SOAP body

The SOAP body is the area where Web services place the actual application data. The SOAP specification does not define the way the body is structured; this is up to the application. However, SOAP contains mapping conventions, which may be used as a hint as to how the message is structured. These mapping conventions consist of the *interaction style*, and the *encoding* of the message.

It is important to note that although the style and encoding belong to the SOAP specification, the data is only processed by the SOAP endpoints, and thus, the endpoints have to agree on these attributes. It is generally impossible to infer the style and encoding from a SOAP message.

SOAP supports two different interaction styles: *document style* and *rpc style.*

- **Document Style**. Document style implies that complete (mostly XML-based) *business documents* are carried in the SOAP body.
- **RPC Style**. RPC stands for *Remote Procedure Call* and implies that the calls taking place are in fact distributed procedure calls. When using RPC, the SOAP body contains either the call itself (i.e., operation name and parameters), or the answer to a call (i.e., the result).

As SOAP is XML-based, the data inside a SOAP body must be formatted in some XML dialect. An *encoding* defines how to encode the actual application data into an XML format. An encoding is identified by an URI and is extensible, but two encoding formats have been specified as part of the SOAP standard:

- **Literal Encoding**. Literal encoding basically means no encoding – the data is just copied literally (and therefore must already be XML-based, or mapped by other means).
- **SOAP Encoding**. The SOAP encoding is a specific encoding format detailed in the SOAP binding specification, describing how to convert structures like arrays, structured data types, and also simple types like `double` or `string` into XML.

Note that in principle, styles and encodings may be mixed arbitrarily, resulting in the following four possible combinations:

- **document/literal**: A very common style, which will, arguably, become the dominant style for Web services.
- **document/encoded**: Not used.
- **rpc/encoded**: Typical style of older Web services, follows the RPC idea.
- **rpc/literal**: Basically, this is the document/literal style, but with additional wrappers for an operation and its parameters.

The different styles and encodings of SOAP have led to much confusion and are among the top reasons hindering interoperability. See section 2.2.4 for more information.

During normal operation, everything inside the SOAP body belongs entirely to the application. However, there is one important exception: Processing errors which may occur in any of the intermediary nodes or the ultimate receiving node necessitate a premature answer to the initial sender. If such an error occurs, a SOAP *fault* element is placed in the SOAP body, which is defined with an *error code*, a *reason* for the error, and possible *application-specific details* about the error.

The SOAP specification contains five SOAP fault codes; one of these must be included in the *value* sub element of the *code* element:

- **VersionMismatch**: Wrong SOAP version.
- **MustUnderstand**: A header was tagged with the `mustUnderstand` tag, but the intended processing node did not understand the header.
- **DataEncodingUnknown**: The specified encoding was unknown to the processing node.
- **Sender**: The message was not formed correctly.
- **Receiver**: The receiving node could not process the message.

The reason sub element of the fault should include a human-readable description of the problem; possibly in multiple languages. The application-specific details belong entirely to the application, as the complete SOAP body would in normal circumstances.

Figure 4 contains an example of a SOAP fault. It includes a fault code (`Sender`), a fault reason, and a detail block with an application-specific fault node.

```
<env:Fault>
     <env:Code>
          <env:Value>env:Sender</env:Value>
     </env:Code>
     <env:Reason>
          <env:Text>Processing error</env:Text>
     </env:Reason>
     <env:Detail>
          <e:myFaultDetails xmlns:e="..." />
     </env:Detail>
</env:Fault>
```

Figure 4: A SOAP Fault

2.2.1.4 SOAP transport protocol bindings

The previous sections described the basic building blocks of a SOAP message, which include a SOAP header and a SOAP body, and information on the processing model, which defines actions to be carried out by SOAP processing nodes based on header information.

However, the SOAP messages must still – somehow – find their way to the target. This task is carried out by a transport protocol, like HTTP or SMTP. The SOAP specification contains information about how to *bind* SOAP to HTTP and SMTP; however, with the right binding specification, any other protocol may be used as well.

It is important to note that the actual target of a message – i.e., the address – is not part of a SOAP message. Although it might be introduced with a specialized header – and in fact, this is done by WS-Addressing – the basic addressing is done by the transport protocol, and the target address – an URL in case of HTTP, or an email address in case of SMTP – must be specified outside of the SOAP message.

The SOAP transport bindings define how a concrete message exchange looks like. These are of course highly protocol dependent. For example, the HTTP binding defines that…

- …in a request-response operation, the request is sent via HTTP POST, and the server replies with a return message and a “200 OK” status code.
- …if a fault occurs, the status code must be “500 Internal Server Error”, and the response must carry the fault information in its payload.
- …if a SOAP message is placed in a HTTP response, the HTTP content header must carry the string `application/soap+xml`.
- …and so on.

The description of the SOAP transport protocol bindings completes this section.

2.2.2 Web Service Description Language (WSDL)

The Web Service Description Language is an XML-based language used to describe Web services – what they can do, where they are located, and what kind of data they expect in which format. WSDL thus combines features commonly found in Interface Definition Languages (IDLs) with access and location information (W3C 2001). This section discusses WSDL 1.1, as WSDL 2.0 was not officially released at the time of this writing.

WSDL is a hybrid language created by IBM and Microsoft by combining their former proprietary service description languages – IBM’s Network Accessible Service Specification Language (NASSL) and Microsoft’s Service Description Language (SDL). WSDL contains the possibility for describing remote procedure calls (RPC) and using any kind of protocol and data format, taken from NASSL, and the possibility for using messaging, taken from SDL.

WSDL is based on seven core principles (Weerawarana et al. 2005):

- **Extensibility**. WSDL itself only allows a description of the message formats, message interaction patterns, wire format of messages, and the location of the service. However, when describing a service, one could think of many more attributes to attach to the service description – for example, the cost of the service or quality of service agreements. WSDL allows specification of such attributes through extensions, which may be defined and *plugged in* into many of WSDL’s core structures.
- **Support for Multiple Type Systems**. Although Web services are fundamentally based on XML and XML Schema is the dominant type system for Web service messaging, WSDL allows specification of the Web service data in arbitrary type systems, for example in MIME or even platform-dependent type systems.
- **Unifying Messaging and RPC**. Messaging and RPC are two different possibilities of approaching the integration problem. As WSDL was created by merging two languages which have each taken one of these approaches, the resulting language now includes the possibility for using both styles.

- **Separation of “What” from “How” and “Where”**. WSDL documents are split into two parts: An *abstract* part, describing what the service does in general terms (the data format, and available message patterns), and a *concrete* part, describing where the service is located and how exactly to call it (i.e., an URI with the location, and the specification of a wire format). This separation allows offering a service at different locations and using different invocation methods.

- **Support for Multiple Protocols and Transports**. The concrete wire format – or protocol – of the messages or calls sent to a Web service is not dictated by WSDL. Although most Web services use SOAP as the protocol, WSDL allows specification of arbitrary wire formats. Along the same lines, the concrete transport used (for example, HTTP or SMTP) is also arbitrary.

- **No Ordering**. WSDL does not allow specification of an interaction protocol with the service – i.e., the order in which operations must be called. The ordering is part of the semantics of the service (see next principle).

- **No Semantics**. WSDL does not describe any semantics of the services. If the service offers an operation with the name `getResult` and returning an `int`, the actual meaning of this operation must be specified in a separate document, for example as human-readable text, or with the help of semantic description languages like OWL-S (DAML 2004).

As can be seen from the given principles, most of the definitions of WSDL are extensible in some way or another. This has led to a multitude of possible service descriptions, which in turn lower the interoperability of the Web services architecture. This will be further discussed in section 2.2.4.

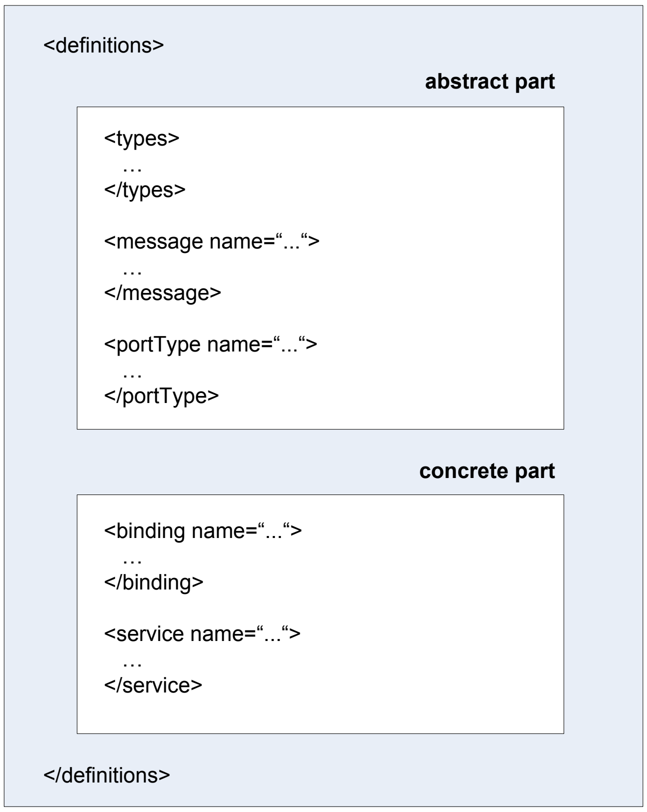


Figure 5: WSDL Document Format

As pointed out above, WSDL documents are written in XML and consist of two parts: An abstract part, and a concrete part. As can be seen in Figure 5, both parts are enclosed in a global `definitions` block. This block contains a few attributes, among them the `targetNamespace` 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, as elements 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.

The `types` element contains the basic data structure definitions of the service, which are later referenced in the `message` elements to define the data to be exchanged. As pointed out above, WSDL supports multiple type systems for specifying the types of these structures. A shortcut is provided for specifying types directly inside the `types` element in XML Schema, but by using the extensibility mechanism of WSDL, any type system is possible, including non-XML based systems.

It is important to note that the data definitions in the types block are, in most cases, only an intermediary format. The actual wire data – mostly formatted in SOAP – will be different from these specifications, as will the data handled internally by the Web service, which is dependent on the concrete implementation of the service (for example, if the service is implemented in Java, the Java type system is used). Still, the data structures defined in the types block are *the closest thing to a native data format* of a Web service.

Figure 6 shows an example of an inline XML Schema definition inside a types block (taken from W3C 2001). It defines two elements with complex types for later use: `TradePriceRequest`, which contains a single element of the basic type `string`, and `TradePrice`, which contains a single element of the basic type `float`.

```
<types>
   <schema targetNamespace="http://example.com/stockquote.xsd"
           xmlns="http://www.w3.org/2000/10/XMLSchema">
       <element name="TradePriceRequest">
          <complexType>
              <all>
                  <element name="tickerSymbol" type="string"/>
              </all>
          </complexType>
       </element>
       <element name="TradePrice">
          <complexType>
              <all>
                  <element name="price" type="float"/>
              </all>
          </complexType>
       </element>
   </schema>
</types>
```

Figure 6: WSDL Types Section

After having defined the data structures, the messages used for exchanging data with the Web service are specified. A WSDL message is the basic data package which may be sent to the Web service, or is sent by the Web service.

Messages are specified by means of the `message` construct. Each message has a name by which it is later referenced, and consists of a number of parts to define the data to be transferred. Each part has a type, which is either taken from the `types` section, or is a basic type from the type system in use. To allow this kind of specification, the `message` construct is extensible, too, allowing plug-ins for specifying types of parts in arbitrary type systems.

Figure 7 shows two examples of message specifications. Each message has only one part, which has an *element type* taken from the types section defined in Figure 6.

```
<message name="GetLastTradePriceInput">
    <part name="body" element="xsd1:TradePriceRequest"/>
</message>

<message name="GetLastTradePriceOutput">
    <part name="body" element="xsd1:TradePrice"/>
</message>
```

Figure 7: WSDL Message Section

In the example, the `element` attribute is used for specifying the type, which references an XML Schema element. Another option would be the `type` attribute, which directly references an XML type. This distinction will later become important when discussing message styles and encodings.

A message definition just specifies a set of data parts; it does not specify how the client of the Web service should handle these messages. Rather, message constructs are referenced in *operations*, which are specified as part of `portTypes`.

A `portType` can be seen as an interface in much the same way as an interface in languages like Java or C#, containing a list of operations which may be invoked on a service, each specifying input, output, and possible fault thrown by the operation. In WSDL, the input, output, and fault elements are specified as messages from the message constructs.

A `portType` has a name and contains an arbitrary number of operations. There are four types of operations in WSDL; the type of an operation is defined by the order of the messages it sends and receives:

- **One-Way**. A one-way operation just receives a message from a client of the service, not returning anything.
- **Request-Response**. A request-response operation is the typical Web service interaction. A message is received from a client, and an answer is generated and sent back to the client. This operation may additionally specify fault elements, which are returned to the client instead of the answer if an exception occurs.
- **Solicit-Response**. In a solicit-response operation, the service itself initiates the operation by sending out a message, and expects a message in return. This operation may also specify fault elements.
- **Notification**. The service sends a message, and does not expect an answer.

The first two operations are sometimes called *inbound* operations, as they are initiated by a client of the service, whereas the second two operations are called *outbound* operations, as they are initiated by the service itself (Weerawarana et al. 2005). Although specified as part of the WSDL standard, the outbound operations cannot be properly used as no bindings exist for these kinds of operations.

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 2.2.3 offers the necessary infrastructure for such callbacks.

Figure 8 shows an example of a `portType` definition with one operation, called `GetLastTradePrice`. The first element in the operation is an input element, and the second one an output element, rendering this operation a request-response operation. The input element references the `GetLastTradePriceInput` message from Figure 7; the output element references the `GetLastTradePriceOutput` message from the same figure.

```
<portType name="StockQuotePortType">

    <operation name="GetLastTradePrice">
        <input message="tns:GetLastTradePriceInput"/>
        <output message="tns:GetLastTradePriceOutput"/>
    </operation>

</portType>
```

Figure 8: WSDL Port Type Definition

The specification of `portTypes` completes the abstract part of a WSDL document. With types, messages, and port types specified, the "*what*" part is complete: A client knows which operations are available, which messages with which parts must be sent in order to invoke an operation, and which messages with which parts are to be expected back (if any).

However, the description is still abstract in the sense that the actual location of the service and the wire format of the messages – the "*where*" and "*how*" parts – are yet unspecified. This information is contained in the concrete part of a WSDL service description, which may contain an arbitrary number of `binding` and `service` elements.

A port type, describing a group of operations, must be *bound* to a specific data exchange format and a transport protocol in order to be used by a client. To bind a port type to a format, the `binding` element is used. A binding specifies how the data to be used in an operation – specified as a part in a message with a certain type – should be formatted to be sent on the wire, and what transport protocol to use for sending the data.

As WSDL does not assume any specific format nor transport protocol, the `binding` element is a mere wrapper around arbitrary binding specifications, which might include anything from SOAP and HTTP to non-XML based formats using a proprietary network protocol. Having said that, the dominant binding for Web services is of course the SOAP format and the HTTP transport, which are in fact, although declared optional, part of the WSDL specification.

The SOAP binding defines how a SOAP message is created from an input, output, or fault message element of a WSDL operation, and also how to unwrap the data from a SOAP envelope on the other side. How the data – which can take arbitrary form, as pointed out when discussing the `types` section – is placed inside the SOAP body is defined by two parameters of the SOAP binding, namely the `style` and `use` attributes, which directly correspond to the SOAP style and encoding, discussed in section 2.2.1. WSDL leverages the SOAP *style* to provide both RPC and messaging:

- **Messaging**: By specifying `document` as the operation style, all the parts of a message are basically directly inserted into the SOAP envelope. Note that in document style, the parts of the message itself, including their names, do not play an important role – mostly, there is only one part. In document style, part types are normally specified as *elements*, and these elements form the top-level entry points into the data.

- **RPC**: By specifying `rpc` as the operation style, the parts of a message are wrapped into an outer XML element representing the remote procedure call. The result is then inserted into the SOAP envelope. In rpc style, the parts of the message represent the named parameters of the call and are thus very important; they are normally specified as *types* and form the top-level entry points into the data themselves, directly containing the data.

The `use` attribute directly corresponds to the SOAP encoding as discussed in section 2.2.1, specifying how a concrete implementation of a Web service client must encode the data to be sent to this Web service.

Figure 9 shows an example of a binding specification which defines a binding for the `StockQuotePortType` defined in Figure 8. The binding is defined using the SOAP binding extension, defining a document/literal style of operation and the use of the HTTP transport.

```
<binding
     name="StockQuoteSoapBinding"
     type="tns:StockQuotePortType">

     <soap:binding
          style="document"
          transport="http://schemas.xmlsoap.org/soap/http"/>

     <operation name="GetLastTradePrice">

          <soap:operation
               soapAction="http://.../GetLastTradePrice"/>

          <input>
               <soap:body use="literal"/>
          </input>
          <output>
               <soap:body use="literal"/>
          </output>

     </operation>

</binding>
```

Figure 9: WSDL Binding Specification

Having defined bindings for all `portTypes`, these bindings are used to define `ports`, which integrate the binding specification (i.e., the concrete format of the Web service calls) with the actual address of the service. The specification of the address is extensible as well.

Ports are then grouped into services. A `service` can, finally, be seen as the complete representation of a Web service, specifying which ports are available at which address, and how to invoke the operations inside the referenced `portType` by means of the given binding.

Figure 10 shows an example of service definition, using the binding specified in Figure 9, and a HTTP address specified by means of the SOAP extension.

```
<service name="StockQuoteService">

    <port name="StockQuotePort" binding="tns:StockQuoteBinding">
        <soap:address location="http://example.com/stockquote"/>
    </port>

</service>
```

Figure 10: WSDL Service Description

With the description of the `service` element, the concrete part of a WSDL document is complete, thus also completing the explanation of WSDL itself.

2.2.3 Web Service Addressing (WS-Addressing)

As already noted in the previous sections, routing messages to the correct Web service and, in the case of asynchronous callbacks, back to the original sender (or a different Web service, for the matter) requires additional addressing information to be included in the exchanged messages. The Web Service Addressing standard (WS-Addressing, W3C 2006a) provides such means by defining the notion of a *service endpoint* and how to *encode addressing information* in a message (specifically, a SOAP message).

The WS-Addressing specification was designed to deal with the following three characteristics of Web service communications (Weerawarana et al. 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, for example when a message is first routed within a company, and then between enterprises over the Internet.
- **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 provide a solution for all these requirements, WS-Addressing provides a protocol-independent and 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. Also, although WS-Addressing is part of the messaging layer and thus one layer beneath service description, it contains strong ties to WSDL.

2.2.3.1 Defining endpoint references

An endpoint reference is defined as a data structure which contains all the necessary data to identify a service endpoint at runtime; a service endpoint being defined as any potential source or destination of Web service messages.

An endpoint reference contains two kinds of data: *Runtime information*, which is required for actually interacting with the endpoint, and *metadata*, which is intended for the recipient of the message and provides more information about the intent of the message.

The runtime information of an endpoint consists of the following three parts:

- **Address property**. The address property is the only mandatory component of a service endpoint, and contains the address of the endpoint in the format of whatever protocol is used for communication.
- **Reference properties**. The reference properties contain additional information about the location of the endpoint. Their values depend on the concrete protocol and environment.
- **Reference parameters**. Reference parameters contain information which is not relevant for addressing and identifying the endpoint, but provides details for a successful interaction with the endpoint.

The address field and reference properties together form the *ID* of the endpoint, i.e. every message which contains the exact same values in these two parts is targeted at one identical endpoint.

The metadata associated with an endpoint reference helps applications to configure their interactions with the endpoint. The metadata contains three fields:

- **WSDL service name**. The WSDL service name may be added to identify the static representation of the service.
- **WSDL port type**. In addition to the service name, this field identifies the port to be used.
- **Policies**. This field may include policies as defined in the WS-Policy standard (for more information, see IBM et al. 2006).

Figure 11 shows an example of an endpoint reference which contains an address property and both a reference properties and parameters block, each carrying one element.

```
<wsa:EndpointReference xmlns:wsa="..." xmlns:fabrikam="...">

   <wsa:Address>http://www.fabrikam123.example/acct</wsa:Address>

   <wsa:ReferenceProperties>
       <fabrikam:CustomerKey>123456789</fabrikam:CustomerKey>
   </wsa:ReferenceProperties>

   <wsa:ReferenceParameters>
       <fabrikam:ShoppingCart>ABCDEFG</fabrikam:ShoppingCart>
   </wsa:ReferenceParameters>

</wsa:EndpointReference>
```

Figure 11: WS-Addressing Endpoint Reference

2.2.3.2 Message information headers

WS-Addressing information is intended to be included in the headers of whatever messaging protocol is in use. As WS-Addressing is part of the Web service standards family, the protocol will be mostly SOAP, and a binding for SOAP is included in the specification and will be presented in section 2.2.3.3.

For the purpose of identifying the target of the message and adding metadata, WS-Addressing defines *message headers*. The concrete serialization of the headers, i.e. how they appear in the final message, depends on the messaging protocol.

Two headers are required in every message (regardless whether one-way or request-response):

- **“To” header**. This header contains the address of the target endpoint. For example, if HTTP is used, this header contains an URL.
- **“Action” header**. This header contains an “identification URI”, defining the semantics of the message.

The identification URI of the action header can take any form; when considering WSDL-based messaging, it will point – in some way – to the actual operation which is invoked with this message.

Being part of the Web service standards family, WS-Addressing defines mappings between WSDL operations and the action header. The mapping is either explicit or implicit:

- **Explicit mapping**. When using the explicit mapping, the WSDL of the service must be augmented – via the extension mechanism – with information for WS-Addressing. Specifically, each input, output, or fault of an operation is tagged with an URI to be used in the WS-Addressing action header. Figure 12 shows an example of such augmentation.
- **Implicit mapping**. If an explicit mapping is impossible or undesired, the value of the action URI is derived by concatenating the target namespace URI of the WSDL definition with the port type and operation names, separated by slashes.

```
<portType name="StockQuotePortType">

     <operation name="GetLastTradePrice">

          <input message="tns:GetTradePricesInput"
               wsa:Action="http://example.com/GetQuote"/>

          <output message="tns:GetTradePricesOutput"
               wsa:Action="http://example.com/Quote"/>

   </operation>

</portType>
```

Figure 12: WS-Addressing WSDL Port Type Extension

One major application area of WS-Addressing is to enable callback addressing in (possibly asynchronous) request-response operations. For this purpose, five headers have been defined:

- **“ReplyTo” header**. The *ReplyTo* header contains a complete endpoint reference to send the reply message to. It must be present in the first message of a request-reply operation.
- **“FaultTo” header**. The optional *FaultTo* header specifies an endpoint reference to send faults to. If it is not present, the *ReplyTo* header is also used for faults.
- **“Source” header**. The optional *Source* header specifies the endpoint which originally sent the message. This may be useful for correlation if *ReplyTo* points to another endpoint.
- **“MessageID” header**. This header contains an ID which uniquely identifies the message which it is part of. Specifying a message ID allows creating relationships between messages. This header is required in the first message of a request-reply operation.
- **“RelatesTo” header**. This header is the counterpart of the *MessageID* header and must contain the *MessageID* of the previous message in a message exchange. The *RelatesTo* header has an attribute specifying the communication relationship with a code (the *RelationShipType*), which is defined in the WS-Addressing specification. For example, a reply in a request-response operation carries the attribute “*Reply*”, which is also the default value if no attribute is specified.

To illustrate the use of these headers, the next section will contain a complete example of an asynchronous request-response operation using the SOAP binding of WS-Addressing.

2.2.3.3 A SOAP-based asynchronous request-response operation

To enable an asynchronous callback, a request message must carry at least a message ID identifying the message, and a reply-to block which includes the information where to send the callback, i.e., an address. Additionally, the request must contain the addressing information for the request itself.

Figure 13 shows an example of WS-Addressing request headers being included in a SOAP header. The concrete protocol used for the exchange is SMTP.

```
<S:Envelope xmlns:S="http://www.w3.org/2003/05/soap-envelope"
     xmlns:wsa="http://www.w3.org/2005/08/addressing">
     <S:Header>

       <wsa:MessageID>http://example.com/uniquestring</wsa:MessageID>

       <wsa:ReplyTo>
          <wsa:Address>http://example.com/bclient1</wsa:Address>
       </wsa:ReplyTo>

       <wsa:To>mailto:fabrikam@example.com</wsa:To>
       <wsa:Action>http://example.com/fabrikam/Delete</wsa:Action>

     </S:Header>
     <S:Body>
        ...
     </S:Body>
</S:Envelope>
```

Figure 13: WS-Addressing SOAP Request Message

The receiver of this message must follow three steps to create a reply message:

- The *ReplyTo* header must be read, and the reply address extracted and used for addressing the reply message. The address must also be included in the "*To*" *header* of the returning message.
- The *MessageID* header must be read and included in a *RelatesTo* header in the returning message.
- The *RelatesTo* header of the returning message must be set to carry the *RelationShipType* "*Reply*".

The resulting message looks like the one in Figure 14. The message ID of the incoming message has been copied over to the *RelatesTo* header. Note that the *RelatesTo* header does not carry an attribute of type "*Reply*", as this is already the default. Also, the message has a new, own message ID for further communication, and its own set of "*To*" and "*Action*" headers.

```
<S:Envelope
     xmlns:S="http://www.w3.org/2003/05/soap-envelope"
     xmlns:wsa="http://www.w3.org/2005/08/addressing">
     <S:Header>

       <wsa:MessageID>http://example.com/uniquestring2</wsa:MessageID>

       <wsa:RelatesTo>http://example.com/uniquestring</wsa:RelatesTo>

       <wsa:To>http://example.com/business/client1</wsa:To>

       <wsa:Action>http://example.com/mail/DeleteAck</wsa:Action>

     </S:Header>
     <S:Body>
          ...
     </S:Body>
</S:Envelope>
```

Figure 14: WS-Addressing SOAP Response Message

The return message contains an URL as the *To* address, not an E-Mail address. This message is thus sent via HTTP.

This example completes the description of WS-Addressing.

2.2.4 A critical outlook on Web services

The Web service family of standards, and the technology itself, are fairly new. The three standards which form the basis of the Web service framework – SOAP, WSDL, and UDDI, two of which have been presented in the previous sections – have been made public in their initial versions in mid to late 2000. Since then, SOAP has moved to version 1.2 and WSDL to 1.1. WS-Addressing 1.0 was presented as recently as May 2006.

Despite this, Web services are seemingly regarded by many as a silver bullet for solving the problems of enterprise computing, which generates a substantial hype around the technology. On the other hand, it is increasingly becoming clear that when looking beyond the marketing front pages, there is still much work to be done.

The main technological point of criticism when talking about Web services is *interoperability*. Web services have been designed – or so it is claimed – for inter-enterprise application, spanning different vendors, operating systems, and development platforms. To accommodate many uses, the standards have been designed in an open and extensible way, allowing users of Web service technology to re-use their existing solutions by plugging them into the appropriate extension points of the standards.

However, extensibility has its drawbacks, too. Every software system implementing a Web service middleware – i.e., software which reads and writes WSDL, creates and sends SOAP messages, or routes messages according to WS-Addressing information – must be able to deal with *all the extensions used*, or fail in the process of handling a Web service artifact. When regarding the first definition of a Web service of W3C (see section 2.2), WSDL and SOAP are themselves optional. However, even if WSDL and SOAP are required as the underlying standards, there are still many points for extensions; for example:

- The underlying type system of a WSDL Web service can be freely chosen. Mostly XML Schema is used, but any type system is possible.
- The transport protocol specified in the SOAP binding, which is used to talk to a Web service, can be freely chosen. Mostly, HTTP is used, but the possibilities are, again, unlimited.
- The format (style and encoding) of the data inside the SOAP body can take any form. The two styles (document and rpc) and encodings (literal and encoded) defined as part of the SOAP standard are not the only choices.
- The mechanism for adding callback information to a SOAP envelope for creating callbacks is user-defined. WS-Addressing is only one possibility.

These are just some examples of the possibilities of extending WSDL and SOAP to a point where nearly nothing is left of the original standards. However, even when every standard mechanism is used – WSDL with XML Schema as a type system, the SOAP binding, WS-Addressing for adding callback information – creating interoperable Web services is still very hard. This has to with three issues:

- First of all, the standards have significant flaws, i.e. can be interpreted in different ways, which leads to different implementations by different vendors. One major area of problems is the style and encoding of SOAP messages, which have led to much interoperability trouble.
- Secondly, as pointed out above, the standards are still quite young. They change significantly from version to version, creating problems when different versions are implemented.
- Thirdly, there is the problem of converting native data (for example, from Java) into XML and back, which is a non-trivial task as well, especially if there are semantic assumptions which cannot be properly expressed in an XML Schema.

Most of these problems not only affect vendors of Web service middleware systems who implement the Web service functionality, but also the users working with these platforms, as the problems "leak" through to the top as soon as one tries to interoperate with other implementations, languages, or operating systems – and sometimes even when staying within one platform.

The industry has recognized that these Web service interoperability problems require some sort of action, and has formed the Web Service Interoperability Organization (WS-I) to find a remedy for the situation. The WS-I is backed by major industry players like IBM, Microsoft, Sun, and Oracle, and intends to "*promote Web services interoperability across platforms, operating systems and programming languages*" (WS-I 2006b).

The most important publication of WS-I is the **WS-I Basic Profile 1.1** (WS-I 2006a), which was released in April 2006. A profile, as WS-I states, provides "*implementation guidelines for how related Web services specifications should be used together for best interoperability*".

The Basic Profile is a sort of "Über-Specification", covering among others the WSDL and SOAP specifications, and containing a formal clarification of many ambiguities, inconsistencies, and errors in the WSDL and SOAP standards. Some important statements of the Basic Profile include:

- A Web service must be described in WSDL 1.1.
- The only binding allowed is SOAP, and the only transport to be used with the binding is HTTP.
- The only operations allowed are One-Way, and Request-Response.
- The style and encoding used must be either rpc/literal or document/literal (in particular, the SOAP encoding may no longer be used).

The changes introduced by the WS-I Basic Profile offer major relief for implementers of Web service technology, as not only many of the errors and inconsistencies were dealt with, but also many of the extension points closed down, only allowing certain, well-known extension specifications.

However, many of the changes necessary for WS-I Basic Profile conformity are of a breaking nature. Thus, although backed by major industry players, it is expected that the introduction of these changes will take quite some time and may even never be fully accomplished. Nevertheless, the Basic Profile is huge step forward for Web service interoperability.

To summarize, Web service interoperability is still a major problem for implementers of Web service technology, as many existing software applications and Web services still use now-outdated elements of the basic Web service standards. It is hoped that the WS-I Basic Profile will gain industry-wide respect, thus leading to less diverse service descriptions and messaging requirements.

2.3 Service composition with BPEL

The ability to integrate, or *compose*, existing services into new services is, arguably, the most important functionality provided by SOAs (Juric 2006). At its very core, composition of services should allow creating, running, adapting, and maintaining services which rely on other services in an effective and efficient way (Stojanovic et al. 2005).

Environments which support service composition have their roots in workflow management, where business logic was implemented by composing existing, coarse-grained applications (Alonso et al. 2004). SOAs, especially when using the Web service implementation, take composition to the next level by offering a much more homogenous environment, as all parts of a composition are services themselves, (ideally) described in the same fashion, and communicating with the same messaging standards. Service composition differs from traditional composition approaches in two important ways:

- **Service composition is recursive**. A composition of services is, in most cases, a service itself, which can be composed even further. This is an elegant way of dealing with complexity in large service-based systems.
- **Service composition works in a distributed fashion**. In traditional composition approaches, components are compiled, linked, included, and sold with the final composition. Service compositions, on the other hand, use existing services as-is, invoking them via a network.

One approach to service composition is using conventional languages, for example existing object oriented languages like C# or Java. However, most of these languages were not designed with service composition in mind – they do not regard services as first-level entities. A more natural approach to service composition is enabled by high-level composition environments and tools, in which the developer never leaves the service paradigm, i.e. is able to deal with services directly instead of their underlying implementation principles.

Service composition is an implementation technology, and is enabled by a composition middleware which consists of three parts (Alonso et al. 2004):

- **Composition model and language**. Composition of services means creating a workflow, or *composition program*, which defines the order in which the composed services are invoked, how data is transferred, and other implementation logic. A composition model provides the background for the design of the composition, and a language is provided for actually creating the composition program.
- **Development environment**. The development environment consists of an editor for the composition language, and can be seen as an IDE for the service composition model.
- **Runtime environment**. A service composition must be executed, thereby creating *composition instances* which follow the predefined operation pattern of the composition program.

There are two distinct ways for designing composite services, which have been named *choreography* and *orchestration* (Stojanovic et al. 2005):

- **Choreography**. A choreography describes a collaboration between services to achieve a certain goal. The control logic of a choreography is distributed – each service knows what to do and whom to contact; these actions are not described as part of the choreography. A choreography is an *abstract* definition of an interaction, intended to convey the general purpose and goal of the composition.
- **Orchestration**. An orchestration, on the other hand, focuses on one service, specifying the concrete actions to take to implement that service by using other services. The control logic is therefore centralized in the orchestration. An orchestration is intended to be executed.

However, the line between choreography and orchestration is beginning to blur (Juric 2006), and in fact some languages now offer a common subset of operations for both approaches.

As outlined in previous sections, many hopes about SOAs and Web services involve a supposed facilitation of enterprise computing, and an efficient and flexible modeling of *business processes*. Service composition, especially orchestration, is a means of achieving the goal of creating a business process out of services. In fact, Juric 2006 defines a business process as follows:

A business process is a collection of coordinated service invocations and related activities that produce a business result, either within a single organization or across several (Juric 2006).

The Business Process Execution Language (BPEL), which already carries all necessary ingredients in its name, has been created with the purpose of achieving this goal, and is presented in the next section.

2.3.1 An introduction to BPEL

The Business Process Execution Language (BPEL, Andrews et al. 2003) is an XML-based language which allows defining both choreographies and orchestrations of Web services. BPEL is based on WSDL, XPath, and XML Schema – it uses services described in WSDL for composition, offers the compositions themselves as WSDL Web services, and allows handling of XML Schema based data with XPath expressions.

BPEL has been designed to explicitly address composition requirements in Web service environments, i.e. environments which consist of loosely coupled, distributed, WSDL-based entities which communicate using XML. The design of BPEL fulfills the following requirements (Weerawarana et al. 2005):

- **Flexible integration**. As BPEL is intended to be used to create business processes, it must be flexible enough to accommodate all kinds of interaction patterns and business scenarios in a way which also allows rapid changes.

- **Recursive composition**. As pointed out above, SOAs offer the ability for recursive composition, and BPEL should pass this benefit on to SOA users.
- **Separation of composeability from other concerns**. The composition logic itself should be decoupled from other concerns like coordination protocols, quality of service, and other policies.
- **Stateful conversations and lifecycle management**. Business processes are potentially long-running and may have state. BPEL should support the creation of such processes by using a defined lifecycle management for compositions.
- **Recoverability**. Expected and unexpected errors in business processes must be dealt with by fault handling and compensation mechanisms.

BPEL was created in 2002 by IBM, BEA, and Microsoft, who merged their existing standards WSFL (Web Services Flow Language (Leymann 2001), by IBM) and XLANG (Thatte 2001, by Microsoft). A revised edition with input from SAP and Siebel was released in March 2003 as BPEL 1.1. This version was also submitted to OASIS in April 2003 for standardization. OASIS has been working on BPEL 2.0 ever since.

WSFL describes a composition as a directed, acyclic graph of *activities*; the edges are called *control connectors*. Both nodes and edges may be annotated with transition conditions which allow controlling the flow of the graph. WSFL also introduces the notion of *partners*, describing the relationship of the composition with other services.

XLANG, on the other hand, takes a more structured approach by providing constructs for *sequential* and *parallel control flows*, which then contain activities that are executed according to the enclosing construct. XLANG also introduces *compensation*, custom *correlation* of messages, *exception handling*, and *dynamic service endpoints handling*.

As BPEL is a combination of these two languages, it is possible to use both a graph-oriented and a structured approach when programming in BPEL. Both may also be used simultaneously in a BPEL program.

Although the name BPEL already contains the word *execution*, the BPEL language supports both choreography and orchestration. Depending on which paradigm is used, the created processes have different names:

- **Executable business processes** follow the orchestration paradigm. They specify the exact details of an interaction and can, as the name implies, be executed by a BPEL engine. This seems to be the dominant use of BPEL (Juric 2006).
- **Abstract business processes** follow the choreography paradigm. They capture the general message flow between parties, but lack interaction details and thus cannot be executed.

As this thesis focuses on BPEL composition testing, only executable business processes are relevant and will be discussed further. For more information on abstract business processes see Andrews et al. 2003.

It is important to note that the BPEL specification only includes the composition model and an XML-based composition language – it does not define a development- or a runtime environment. Although both are required to actually design and run a BPEL process, they are out of scope for the specification and must be provided by third parties. Note that this includes a (possible) graphical notation for BPEL – the specification itself only includes XML code.

Without a serious competitor, BPEL has quickly gained industry-wide acceptance and is now the dominant technology for the (admittedly rather new) market of Web service composition.

2.3.2 Partners of a BPEL process

A BPEL process is intended to compose services. To achieve this goal, these services must be identified and invoked; BPEL requires them to be described in WSDL. Likewise, the BPEL process itself is invoked by another party to kick off the execution: as mentioned above, a BPEL process is a Web service itself and must also be described in WSDL.

A Web service which interacts with the BPEL process in any way is called a *partner*. The relationship of the BPEL process with a partner can take three forms:

- The partner invokes the BPEL process.
- The partner is invoked by the BPEL process.
- Both – the partner is invoked by the BPEL process and invokes the BPEL process.

BPEL does not differentiate between a client (which calls the BPEL process) and a Web service which is invoked by the BPEL process, as an invoked Web service may become a client himself when asynchronous callbacks are used. Instead, the relationship between a partner and the BPEL process is described in an abstract way, defining *roles* which the process, or a partner, might take.

BPEL uses the notion of a *partner link type* to model these relationships. A partner link type contains at least one and at most two *roles* – a role being played by either the process or a partner, which is not specified as part of the partner link type, but later as part of the process definition (see below). As BPEL is based on WSDL, services are expected and provided as *WSDL port types*; thus, a role includes a port type which must be provided by whoever takes on that role.

Partner link types are defined as a WSDL extension; thus, they reside within a WSDL file – for example, the BPEL process WSDL file. If external partners are involved, new WSDL files can be created to wrap the existing partner WSDL files, augmenting the information.

Figure 15 gives an example of a partner link type definition (Andrews et al. 2003):

```
<plnk:partnerLinkType name="invoicingLT">

    <plnk:role name="invoiceService">
        <plnk:portType name="pos:computePricePT"/>
    </plnk:role>

    <plnk:role name="invoiceRequester">
        <plnk:portType name="pos:invoiceCallbackPT"/>
    </plnk:role>

</plnk:partnerLinkType>
```

Figure 15: A PartnerLinkType Definition

The partner link type, named `invoicingLT`, carries two roles, one named `invoiceService`, and the other `invoiceRequester`. Each role is tied to a certain port type. Note that the port type referenced may stem from any referenced service, identified by the namespace.

When defining the actual BPEL process later on, one has to decide which role the BPEL process should take, and which role is to be played by a partner; the resulting relationship is called a *partner link*. When looking at the names of the roles and the port types, it becomes (semantically) clear that the first role offers something (to compute a price), and the second role receives something (it offers a callback port type for receiving the computed price).

There are two possibilities in this case:

- The BPEL process itself computes the price. In this case, the BPEL process plays the first role. The computed price is then returned to the client, which is played by the second role.
- A partner computes the price. In this case, the partner plays the first role. The computed price is returned by the partner via callback to the second role, which is played by the BPEL process.

Note that a partner link type may also only have one role, which may likewise be played by the process or a partner. A partner link with only one role does *not* indicate a one-way message flow; it merely indicates that there is no explicit callback (i.e., re-addressing from the side of the invoked role) involved. The port type may still carry request-response operations.

2.3.3 BPEL process basics

As mentioned above, a BPEL process is a Web service and thus needs a WSDL description. This description must be created first – the WSDL port types and operations defined in the description will then be implemented in the BPEL process. Additionally, partner link types must be defined, which will be used in the BPEL process to define the roles played by the process and partners.

A BPEL process is defined as an XML document. Mostly, the file ending *.bpel* is used, although the simpler *.xml* is also common. The root element of each BPEL process file is the `process` element. Among other things, this element defines the name and target namespace of the BPEL process.

Inside the process element, the business process is specified. It is beyond the scope of this introduction to capture all the intricacies of the BPEL language; thus, the focus will be on some of the major concepts which are necessary to understand the overall flow inside a BPEL process.

The process element contains sections, three of which are of immediate interest: The *partner links section*, the *variables section*, and the *activity section*.

The partner link section has already been hinted at. It contains definitions of partner links, which are based on partner link types, augmented with the information of who is going to play which role in the process. A partner link is used in the BPEL process for invoking other Web services or being invoked by a client; it can be seen as a communication channel. The channel itself is handled by the BPEL runtime engine; the BPEL process deals directly with the operations of the partners or clients by using the partner link.

Figure 16 contains an example of a partner link definition. The partner link is based on the partner link type from Figure 15. The definition specifies that the BPEL process plays the role of the requester (`myRole`), and a partner must play the role of the price calculator (`partnerRole`).

```
<partnerLink
     name="invoicing"
     partnerLinkType="lns:invoicingLT"
     myRole="invoiceRequester"
     partnerRole="invoiceService"
     />
```

Figure 16: A Partner Link Definition

In a one-role partner link type, either `myRole` or `partnerRole` can be specified, depending on who is to play the role.

Two things are important to note here. First of all, the partner link has a name. The name is very important, as it will be used in the remainder of the process to identify this partner link. The partner link models a conversation with a partner or client; the lifetime of this conversation is the lifetime of an entire BPEL process instance.

Secondly, the partner link definition does not specify where to find the partner. As the partner link type definition did neither (it only references an abstract port type), it is still unclear where the partner actually resides. The BPEL specification does *not* include any mechanism for defining this, as the binding of actual partners to the partner links is a deployment issue and must be implemented by BPEL runtime providers. See section 2.3.9 for some of the concepts providers have come up with. For the moment, it can be assumed that the BPEL middleware handles binding partner links to concrete partners.

The second section of interest in the process is the variables section. As a programming language, BPEL needs a place to store instance data, like temporary data used for calculations or data received from or sent to partners.

BPEL variables must be declared before use with a name and type. As BPEL is based on WSDL and XML Schema, it is not surprising that the type of a variable is a WSDL message, an XML Schema element, or an XML Schema simple type. As XML Schema is the dominant type system for WSDL and in fact required by the WS-I Basic Profile, *XML Schema can be seen as the de-facto type system of BPEL*, and any variable data is literal XML which may be validated against an XML Schema (Andrews et al. 2003).

Figure 17 shows an example of variable declarations in BPEL. Three variables are defined; the first with a WSDL message type, the second with an XML element as a type, and the third with an XML Schema simple type.

```
<variable
     name="shippingRequest"
     messageType="lns:shippingRequestMessage"
     />

<variable
     name="tempShippingRequest"
     element="lns:shippingRequestElement"
     />

<variable
     name="tempString"
     type="xs:type"
     />
```

Figure 17: BPEL Variable Declaration

All variables declared in the process variables section are global. However, by using *scopes*, one can also introduce local variables (see section 2.3.4.4). BPEL variables are lexically scoped.

The third and last section of interest in the process is the activity section. This section contains the actual business logic as a series of so-called activities. Activities are the statements of BPEL programs. They will be described in the next section.

There are four additional sections in the process which will be discussed later: The event handler section, the fault handler and compensation sections, and the correlation section. They will be described in sections 2.3.5, 2.3.6, and 2.3.8, respectively.

2.3.4 Writing the business logic

The BPEL business logic consists of a series of activities which are executed in sequence or in parallel. Activities are either basic or structured. Basic activities include the invocation of other Web services or receiving incoming invocations; structured activities are used to define the relationship between activities. Most of these activities are described in this section; advanced special-purpose activities for fault and event handling are discussed in the next sections.

Many activities deal with variables or contents of variables. To be able to manipulate variable data, an XML manipulation language is required. BPEL defaults to XPath 1.0 (W3C 1999), but is extensible for other languages. Activities and XPath expressions for manipulating data are presented after discussing the fundamental basic and structured activities.

2.3.4.1 Basic activities

The most important activities in BPEL processes receive and send messages from and to partners. Three activities are defined to execute this behavior:

- The `receive` activity waits for a message from a partner.
- The `invoke` activity invokes a partner.
- The `reply` activity sends an answer for a message previously received with a `receive` activity.

With these three activities, all messaging scenarios including asynchronous messaging can be implemented.

For example, the BPEL process may need to invoke a synchronous request-response operation of a partner. This can be achieved with the code in Figure 18:

```
<invoke
     partnerLink="shipping"
     portType="lns:shippingPT"
     operation="requestShipping"
     inputVariable="shippingRequest"
     outputVariable="shippingInfo">
     />
```

Figure 18: A Synchronous Invoke

The invoke activity blocks until an answer is received from the invoked partner. The activity has five attributes, which have the following meaning:

- **partnerLink**. Contains the name of the partner link to use. The BPEL runtime engine uses this name to identify the actual partner destination.
- **portType**. The given value denotes the WSDL port type of the target service which contains the operation to invoke.
- **operation**. Denotes the WSDL operation to invoke.
- **input variable.** The input variable contains the data to be sent to the service.
- **output variable.** The output variable is initialized with the reply received from the partner.

By omitting the output variable, the `invoke` activity can be used to call a one-way operation. In this case, the `invoke` activity is non-blocking (as no answer is expected).

Achieving an asynchronous invoke involves creating two activities: An `invoke` activity which triggers the operation at the partner, and a `receive` activity which waits for the callback (Figure 19).

```
<invoke
     partnerLink="invoicing"
     portType="lns:computePricePT"
     operation="sendShippingPrice"
     inputVariable="shippingInfo">
     />

<!-- Do something else -->

<receive
     partnerLink="invoicing"
     portType="lns:invoiceCallbackPT"
     operation="sendInvoice"
     variable="Invoice"
     />
```

Figure 19: An Asynchronous Invoke

The `invoke` and `receive` activities are tied together by using the same partner link. Note that the `invoke` activity is missing the `outputVariable` specification, indicating a one-way message. Note also that the `receive` activity must be executed *after* the `invoke` activity – structured enclosing activities provide the means to handle such relationships.

The two activities use different port types and operations due to the fact that in BPEL, asynchronous operations are implemented as two one-way WSDL operations: The first operation belongs to a certain WSDL port type and, in the end, WSDL service implemented by the *invoked* Web service, and the second operation belongs to a different WSDL port type, and in the end, WSDL service implemented by the *invoking* Web service (called the callback operation).

The BPEL process itself also offers operations which can be called by clients. The callback used in the asynchronous invoke example was already one of these operations. However, the main application area of providing operations is to receive proactive calls from clients. Figure 20 shows an example of how to receive a synchronous call and send an answer.

```
<receive
     partnerLink="purchasing"
     portType="lns:purchaseOrderPT"
     operation="sendPurchaseOrder"
     variable="PO">
     />

<!-- Do something else -->

<reply
     partnerLink="purchasing"
     portType="lns:purchaseOrderPT"
     operation="sendPurchaseOrder"
     variable="Invoice"
     />
```

Figure 20: A Synchronous Receive-Reply

This receive-reply pattern is used in most BPEL processes. Usually, the `receive` activity is used as an entry point into the BPEL process and indicates a call from the client. Then, the BPEL process carries out its logic, and in the end, returns its answer to the client by means of the `reply` activity. Once again, the two activities in Figure 20 use the same partner link to identify the communication channel. Note also that the port type and operation attributes carry the same exact values, identifying the single synchronous WSDL request-response operation used by the two activities.

By omitting the `reply` activity, a one-way receive is implemented.

Last, but not least, it is also possible to provide asynchronous operations to clients. In this case, a `receive` activity is followed by an `invoke` activity, as in the example in Figure 21.

```
<receive
     partnerLink="purchasing"
     portType="lns:purchaseOrderPT"
     operation="sendPurchaseOrder"
     variable="PO">
     />

<!-- Do something else -->

<invoke
     partnerLink="purchasing"
     portType="lns:purchaseOrderCallbackPT"
     operation="receiveInvoice"
     variable="Invoice"
     />
```

Figure 21: An Asynchronous Receive-Invoke

Unlike in the synchronous version, the port types and operations differ in the two activities, as the `invoke` activity represents an asynchronous callback.

With these activity patterns, any communication between Web services can be handled. Note that the activities do not handle addressing, although this is required at least in the asynchronous callbacks. Rather, this is transparently handled by the BPEL execution engine.

Besides activities for handling Web service invocations, BPEL also includes three additional basic activities (Andrews et al. 2003), which are very simplistic and only described shortly:

- **Wait**. The `wait` activity waits for a specified amount of time before continuing.
- **Empty**. The `empty` activity does not do anything.
- **Terminate**. The `terminate` activity ends the current process instance.

2.3.4.2 Structured activities

Structured activities determine the order in which enclosed activities are executed. Structured activities may be nested in arbitrary ways, i.e. include basic or other structured activities. BPEL includes activities for sequential control (`sequence`, `switch`, and `while`), concurrent activity execution (`flow`), and reaction to events (`pick`).

The activities for sequential controls are well-known from conventional structured programming languages. The `sequence` activity represents a normal, sequential flow of execution: All enclosed activities will be executed one after another in lexical order. Most BPEL processes will use a `sequence` as the root activity.

The `switch` activity introduces conditional behavior. It contains a series of ordered `case` branches, each including a branch condition, and an optional `otherwise` branch, which is taken if none of the branch conditions hold true. The case branches are tested in the order in which they occur in the source code.

Figure 22 contains an example of a `switch` activity. There are two case branches and one otherwise branch. Note that the conditions are formulated in XPath; details on XPath functions for handling BPEL variables are presented in the next section.

```
<switch>

     <case condition="bpws:getVariableProperty(stockRes,level) > 100">
          <!-- do something #1 -->
     </case>

     <case condition="bpws:getVariableProperty(stockRes,level) >= 0">
          <!-- do something #2 -->
     </case>

     <otherwise>
          <!-- do something else -->
     </otherwise>

</switch>
```

Figure 22: A Switch Activity

The `while` activity repeats an enclosed activity until a condition no longer evaluates to `true`. Figure 23 shows an example of a `while` activity, which executes an included sequence.

```
<while condition="bpws:getVariableData(orderDetails) > 100">
    <sequence>
      ...
    </sequence>
</while>
```

Figure 23: A While Activity

Support for concurrent activity execution is provided by the `flow` activity. This activity is, arguably, the most powerful construct in the BPEL language. All activities inside a `flow` activity are executed simultaneously once the `flow` activity is executed.

The `flow` activity is the most visible remnant of IBM's WSFL language, which was one of the two languages serving as input to the BPEL specification. To allow for complex concurrency scenarios, the `flow` construct allows the expression of synchronization dependencies between the enclosed activities. In particular, execution of an activity can be dependent on other activities and certain conditions.

The synchronization dependencies are expressed by a two-fold approach. Firstly, a *named link* is defined in the `links` sub element of the flow activity. The link is then *attached to two activities* inside the flow – a source, and a target. As a first consequence, the target activity will only be executed *after* the source activity has finished.

Figure 24 gives an example of a flow activity with two links. The first link, `XtoY`, defines a relationship between the nested sequence X as the source, and the nested sequence Y as the target. Y will thus be executed only after X completes. The second link, `CtoD`, defines a relationship between the receive activity C inside the nested sequence Y and the invoke D at the bottom of the flow. The invoke activity D will thus be executed only after the receive C completes.

```
<flow>
     <links>
          <link name="XtoY"/>
          <link name="CtoD"/>
     </links>

     <sequence name="X">
          <source linkName="XtoY"/>
          <invoke name="A" .../>
          <invoke name="B" .../>
     </sequence>

     <sequence name"Y">
          <target linkName="XtoY"/>
          <receive name="C" ...>
               <source linkName="CtoD"/>
          </receive>
          <invoke name="E" .../>
     </sequence>

     <invoke name="D" ...>
          <target linkName="CtoD"/>
     </invoke>
</flow>
```

Figure 24: A Flow Activity

A single link may only have one source and target, but activities inside the flow may be source and target for multiple links. If an activity is the target of multiple links, it is execute if *at least one* source from one of the links has been executed.

This default link behavior may be overridden by two options: *Transition conditions* and *join conditions*.

A transition condition may optionally be added to the source activity of a link. This transition condition sets the *outgoing link status*, which determines if the link will be taken. If the condition is not present, the transition condition is always `true`, which means the link will always be taken and the target activity executed.

By specifying the `transitionCondition` attribute on a source element, a link will only be taken if the source activity has completed *and* the `transitionCondition` (which must be written in XPath) evaluates to true.

A join condition may optionally be added to the target activity of a link. If the condition is not present, the default is a logical disjunction (logical or) of the link status of all incoming links.

By specifying a `joinCondition` attribute on a target activity, the target activity is only started if the specified `joinCondition` evaluates to true. To check the link status of incoming links, BPEL provides a special XPath function which returns the status of a named link (Andrews et al. 2003).

If a join condition evaluates to `false` – i.e., the target activity is not to be executed – the default BPEL behavior is to throw a join fault (see section 2.3.6 for more on faults), which effectively ends processing of the enclosing flow, which may not always be desired. Another option would be to simply not execute the activity and carry on with everything else, eliminating the path beginning with the activity which is not executed (dead path elimination). Such behavior can be enabled by setting the `suppressJoinFailure` attribute of target activities to `true`.

Last, but not least, the `pick` activity allows local reactions to events. Note that this is a separate concept from scope-wide event handling, as discussed in section 2.3.5. A `pick` activity allows the specification of a set of events. When the activity is executed, it waits for the occurrence of one of these events, and executes the activities specified for this event.

There are two types of events which may be specified:

- **Message Events**. A message event occurs when a certain incoming message is received by the process.
- **Alarm Events**. An alarm goes off after a certain duration, or at a specified time.

A `pick` activity *must* include at least one message event. `Pick` is ideal for dealing with asynchronous operations: If an answer is not received within a specified time, the business process carries on with its logic on another path.

Figure 25 shows an example of a pick activity. The activity waits for a message for the completion of an order. If the order is not completed within a certain time, the timeout is handled by the appropriate means.

```
<pick>
     <onMessage
          partnerLink="buyer"
          portType="orderEntry"
          operation="orderComplete"
          variable="completionDetail">
          <!-- activity to perform order completion -->
     </onMessage>

     <onAlarm for="'P3DT10H'"> <!-- 3 days, 10 hours -->
          <!-- handle timeout for order completion -->
     </onAlarm>
</pick>
```

Figure 25: A Pick Activity

2.3.4.3 Handling data and conditions

As pointed out in the beginning of this section, BPEL stores instance data in named variables of a certain type. Once declared, variables must be initialized and may later be used to transfer data to partners, receive data from partners, or store internal data like counters for loops.

There are three distinct possibilities for how the data is structured inside a variable, depending on the meta-type of the variable:

- **WSDL Message Type**. Such variables are mostly used to receive data from partners or to send data to partners (i.e., as input or output to invoke, receive, and reply operations). They contain message parts as the top level elements.
- **XML elements**. In this case, the variables contain the referenced element as the top level element. Although the element may be typed with any XML Schema type, mostly complex types are used.
- **XML Schema simple types**. These variables directly represent the value, as do variables with simple types in other programming languages (no further structure).

Many of the variables in a BPEL process will thus be complex in the sense that they do not carry a single value (like `true`), but a complete XML data structure. To be able to initialize these variables, or extract single values from them, a language for addressing parts of these data structures is required. In BPEL, the default language for this is XPath 1.0.

Variables are used in many places. Two important ones are assignment activities, in which data is assigned to variables, and conditions of while or switch activities. In both cases, XPath expressions are used to deal with variable data.

Variable assignment in BPEL is handled through the `assign` activity. This activity contains one or more copy commands, which in turn each specify a source (`from`) and destination (`to`). This section is concerned with variables; thus, for the moment, source and destination are defined as follows:

- The source may be a variable, a defined content of a variable, an expression, or a literal value.

- The destination must be a variable, or defined content of a variable.

Addressing the defined content is particularly interesting. BPEL provides two mechanisms for this. Firstly, the part of the message may be specified if the variable holds a message type; and secondly, an XPath query may be specified, identifying the element which is to be copied or replaced.

Figure 26 shows how to use assignments for copying data between variables. The first copy operation simply copies the contents of a whole variable. The second one copies only one part from the message-typed variable `c1` into the variable `address` (which must be properly typed for this). The third copy just extracts the last name from an address by employing XPath; and the last copy uses queries for addressing both the source and target.

The last copy operation is interesting as the target may not yet contain structured data, i.e., the query will not yield a result. The BPEL specification states that a fault must occur in this case. However, many vendors of BPEL editors and runtime engines have decided that the intention of the user is clear in these cases, and the structure is simply *created* as part of the copy operation.

```
<assign>
     <copy>
          <from variable="c1"/>
          <to variable="c2"/>
     </copy>
     <copy>
          <from variable="c1" part="address"/>
          <to variable="address"/>
     </copy>
     <copy>
          <from variable="c4" part="address"
               query="/address/lastName" />
          <to variable="lastName" />
     </copy>
     <copy>
          <from variable="c5" query="/ResultElement/snippet"/>
          <to variable="temp" query="/result/snippet"/>
     </copy>
</assign>
```

Figure 26: Simple Assignments

Besides copying variables, the `from` element also allows specification of expressions and of literal values. Literal values are simply XML fragments which are copied to the target. Expressions, on the other hand, are full-blown XPath statements which are also used in condition checking. They can evaluate to a boolean value, a node, or a node set, depending on the functions used.

To allow such expressions to access BPEL variable data, an XPath function has been added which may be used to extract data from variables. This is important in many cases, for example, to allow updating of a counter variable. The function, named `getVariableData()`, allows specification of a variable, possible message part, and possible query into the variable.

A typical example is presented in Figure 27, which updates a counter variable inside a loop.

```
...
<variable name="counter" type="xsd:integer"/>
...
<assign name="IncreaseCounter">
     <copy>
          <from expression="bpws:getVariableData(counter) + 1"/>
          <to variable="counter"/>
     </copy>
</assign>
```

Figure 27: An Assignment with Expressions

As noted before, XPath expressions may also be used in conditions of `switch` and `while` statements. The following figure has been taken from the previous section and shows an example of a `while` loop.

```
...
<variable name="orderDetails" type="xsd:integer"/>
...
<while condition="bpws:getVariableData(orderDetails) > 100">
    <sequence>
      ...
    </sequence>
</while>
```

Figure 28: Condition in a While Loop

The loop uses an XPath expression which extracts data from a variable, and checks whether the integer-typed data is greater than 100.

2.3.4.4 Scopes

BPEL processes may be further structured by introducing *scopes*. A scope defines the behavioral context for a certain part of the BPEL document. It allows the definition of local variables, correlation sets, and handlers (event, fault, and compensation handlers).

Scope definitions are similar to the definition of the complete process, which has been described in section 2.3.3. Scopes may appear anywhere in the BPEL process, including nested into structured activities. Like the process itself, a scope contains activities.

Figure 29 shows the definition of a scope with a variables section and a structured `flow` activity, which may contain other activities.

```
<scope>

     <variables>
          ...
     </variables>

     <flow>
          ...
     </flow>

</scope>
```

Figure 29: A Scope Definition

2.3.5 Event handling

BPEL processes allow the specification of *event handlers* in the process itself and every scope. An event handler, as the name implies, handles events, which occur asynchronously while the main process logic is running. As a consequence of such an event, arbitrary activities may be executed, just like in the normal flow of the process.

An event handler is active as long as the scope it belongs to is active. In case of a global event handler, i.e. one defined in the process itself, the event handler stays active during the complete lifetime of a business process instance. An event handler cannot start a process instance; the instance must already be running.

The events available for event handling are the same as those used in the `pick` activity (see section 2.3.4.2). As an event handler runs in parallel to the actual business logic, incoming messages can for example be used to query the status of a business process.

Figure 30 shows an example of an event handler which listens to an external message, identified by the `onMessage` construct, which is similar to the `receive` activity. In this case, the message represents cancellation of an order, and the instance is terminated as a result of the message.

```
<eventHandlers>
     <onMessage
          partnerLink="buyer"
          portType="car"
          operation="cancel"
          variable="cancelDetails">

          <terminate/>
     </onMessage>
</eventHandlers>
```

Figure 30: An onMessage Event Handler

Figure 31 shows an example of an event handler with an alarm setting. As pointed out above, an event handler belongs to a scope. Time starts running when the scope is executed, and the alarm event is triggered when the specified amount of time has elapsed.

```
<eventHandlers>
     <onAlarm for="bpws:getVariableData
                   (orderDetails,processDuration)">
          ...
     </onAlarm>
</eventHandlers>
```

Figure 31: An onAlarm Event Handler

2.3.6 Fault handling and compensation

During the execution of a BPEL process, exceptional conditions might arise which must be handled appropriately. An invoked Web service may return a WSDL fault to be handled; the BPEL process itself may need to signal a fault to clients, or an error might occur in the underlying runtime environment.

BPEL processes are intended to model long-running business processes. They combine loosely coupled Web services, rendering it nearly impossible to run a BPEL process in a single, atomic operation. When a fault occurs, the BPEL process may have already invoked several other Web services, initiating processes or storing data in a database, successfully or partially completing certain tasks. This work has to be undone if the process is aborted.

In BPEL, fault-handling behavior is generally attached to the process itself and/or single scopes, which are considered as "*recoverable and reversible units of work*" (Weerawarana et al. 2005). To deal with faults, a *fault handler* may be associated with a scope; to be able to undo the work of a scope, a *compensation handler* may be associated with the scope. As an exception, the `invoke` activity may carry its own inline fault handlers to directly deal with invocation faults.

A fault handler catches faults which are raised in a scope. If a fault occurs, processing of the scope is stopped, and a matching catch block is executed, which may contain arbitrary activities. Mostly, these activities will be targeted at reversing partial, unsuccessful work, possibly propagating the fault to the client if necessary. Faults can originate from many sources:

- An invoked Web service might throw a *WSDL fault*, indicating a business-level problem; or a *low-level fault*, indicating a technical problem with the invocation of the Web service. Business-level faults usually contain additional data which explains the problem.
- The BPEL engine might throw a fault as the result of an abnormal condition. Such conditions include using an uninitialized variable, a join failure, an invalid XPath expression, etc.
- A fault may be thrown explicitly in the business process by means of the `throw` activity.

To programmatically raise a fault, the `throw` activity is used. When a `throw` activity is encountered, normal processing of the scope stops and the fault handler associated with the fault is executed. Figure 32 shows an example of throwing a fault. A fault has an (arbitrary) name, and may be associated with a variable containing more information about the fault.

```
<throw
      xmlns:FLT="http://example.com/faults"
      faultName="FLT:OutOfStock"
      />
```

Figure 32: Throwing a Fault

Note that faults which occur in BPEL *must* be handled within the BPEL process; faults are *not* automatically propagated to clients. To send a fault back to a client, the normal `reply` activity is used with a variable containing a fault, and a WSDL fault name.

The fault handler section of a scope which is used to catch faults can contain an arbitrary number of `catch` blocks and an additional, optional `catchAll` block. A catch block catches a single, identified fault, while the `catchAll` block handles all faults not explicitly addressed by a catch block.

A catch block identifies a fault by means of the *fault name*, and/or the *fault variable data*. Either a fault name or a fault variable must be specified, both are also allowed. BPEL matches faults with catch blocks in read order, i.e. the first matching catch block in the source code is executed.

- **Fault name**. The fault name specifies the name of the fault. When a fault is thrown from within BPEL by means of the `throw` activity, the fault name must match the name used in the `throw` activity. When a fault is thrown by a partner, the fault name must match the name defined in the WSDL operation. Faults thrown by the BPEL runtime have standard names which are listed in the specification (Andrews et al. 2003).
- **Fault variable**. A fault may carry associated fault data. A fault matches a catch block if the type of the fault variable is the same as the type of the fault data associated with the fault (XML Schema type, or WSDL fault message).

Figure 33 shows an example of a fault handler specification with one `catch` block, identifying a fault with both name and variable, and a `catchAll` for all other faults.

```
<faulthandlers>

      <catch faultName="x:foo" faultVariable="bar">
            <empty/>
      </catch>

      <catchAll>
            <empty/>
      </catchAll>

</faulthandlers>
```

Figure 33: Specification of Fault Handlers

Besides a fault handler, a scope may also be associated with a compensation handler. While fault handling deals with recovering from incomplete work, *compensation* is the process of *undoing successful work*. Compensation is used if some parts (scopes) of a BPEL process have already successfully completed, and must be reversed because some later scope failed.

As an example, a business process may book a flight and a hotel. If the flight reservation is successful, but the hotel reservation is not, the flight must be cancelled to restore the original state. Due to the environment BPEL processes run in, communicating with loosely coupled, distributed Web services, and their long-running nature, it is not possible to run them as an atomic operation, and likewise impossible to automatically undo work which may have been completed days ago. Therefore, compensation is user-defined and depends on the concrete problem at hand.

A compensation handler for a scope contains arbitrary activities which undo the regular work of the scope. Regarding the example above, the compensation handler could contact the airline Web service again and cancel the flight.

Figure 34 shows an example of a compensation handler which undoes a purchase by invoking an operation called `cancelPurchase` on the same partner as the regular code.

```
<scope>

     <compensationHandler>
          <invoke
               operation="CancelPurchase"
               ...
          </invoke>
     </compensationHandler>

     <invoke
          operation="DoPurchase"
          ...
     </invoke>

</scope>
```

Figure 34: A Compensation Handler

Compensation handlers can be invoked after a scope has been completed successfully by means of the `compensate` activity. Note that the invocation is only allowed from the fault handlers and compensation handlers of the immediately enclosing scope. An interesting thing to note is that the process as a whole may also have a compensation handler, which undoes the complete business process. This compensation handler must be invoked by platform-specific means.

Figure 35 shows an example of an invocation of a compensation handler from within BPEL.

```
<compensate
      scope="RecordPayment"
      />
```

Figure 35: Compensate Activity

2.3.7 Dynamic partner addressing

It has been noted before that the physical location and invocation parameters of the partners of a BPEL process are handled by the BPEL runtime and must be specified by platform-specific means.

Normally, BPEL process partners are indeed specified at design or deployment time, both by platform-specific means. However, BPEL also supports dynamic assignment of partner endpoints – in this case, the BPEL process itself defines the partner to invoke. To achieve dynamic addressing, a BPEL variable must be constructed or initialized from data passed into the process with a WS-Addressing reference as defined in the WS-Addressing specification. The contents of this variable are then copied into a partner link, which may be invoked subsequently.

The partner endpoint must be represented as an `EndpointReference` XML element, as seen in Figure 11 (on page 19). Such a reference may be copied into a variable, or directly into a partner link, by using the literal form of the assignment activity.

An example can be seen in Figure 36.

```
<assign>
      <copy>
            <from>
                  <EndpointReference xmlns="..."
                        <Address>http://localhost/ServiceA</Address>
                  </EndpointReference>
            </from>

            <to partnerLink="PartnerA" />
      </copy>
</assign>
```

Figure 36: Assigning a Dynamic Partner Link

2.3.8 Process instance lifecycle and correlation

The BPEL language does not contain explicit start and stop activities – instance creation and destruction is implicit by design. An instance is created when a message arrives which matches a `pick` or `receive` activity which has been specially tagged. This is done by setting the activity's `createInstance` attribute to `true` – the default is `false`. A BPEL instance is destroyed when the last activity completes, a global fault is thrown, or the `terminate` activity invoked.

A BPEL process specifies the template for a business process, of which multiple instances may be running at the same time. These instances are completely independent from one another.

Once a BPEL process instance is running, it sends out messages to partners, and expects answers in return. The BPEL middleware must be able to route such messages back to the appropriate instance, which is especially difficult when dealing with asynchronous callbacks. In traditional middleware solutions, an ID token was generated and maintained (added to messages and extracted from messages) by the middleware. BPEL, on the other hand, deals with loosely coupled, distributed Web services not under the control of a specific middleware. Therefore, BPEL introduces the (optional) *correlation* concept to achieve the same thing by using the application data itself.

Correlation allows BPEL designers to decide which parts of the messages sent out to and received from partners constitute the ID of the message exchange, and thus, how to map message exchanges to a concrete business process instance. For example, when dealing with a Web service for ordering items, the Web service might given out an order number as part of the first exchange, which can then be used to correlate subsequent messages with this order number to the same business process instance.

As a BPEL process may deal with many different partners, multiple IDs may need to be created, which nevertheless point to the same process instance. In BPEL, correlation is based on two concepts: *properties* and *correlation sets*.

- A property has a name and type, and is mapped to parts of the messages which might be sent or received as part of the interactions of a BPEL process. A property represents one part of an ID for a message exchange.
- A correlation set consists of multiple properties, and represents the complete ID of an application-level conversation with a process instance. As pointed out above, a BPEL process may have multiple correlation sets for different sets of messages sent and received.

Properties are defined in WSDL by means of the WSDL extension mechanism. A property is first defined with a name and type with the `property` construct, and then *bound* to several parts of messages by using the `propertyAlias` construct. Multiple messages of an exchange – for example, an order message and the returning confirmation – may carry the relevant parts at different locations inside the message. These locations are specified by using XPath expression.

Figure 37 defines a property and an alias for a BPEL process which represents an airline supporting flight booking. The property `flightNo` identifies a flight number (a string) and is mapped to two messages: A message to the client with a flight offer, and a message from the client, which confirms the flight.

```
<bpws:property name="flightNo" type="xsd:string" />

<bpws:propertyAlias
     propertyName="tns:FlightNo"
     messageType="tns:FlightInformationMessage"
     part="flightData"
     query="/flightData/FlightNo" />

<bpws:propertyAlias
     propertyName="tns:FlightNo"
     messageType="tns:FlightConfirmationMessage"
     part="confirmationData"
     query="/confirmationData/FlightNo" />
```

Figure 37: Correlation Properties and Aliases

The so-defined properties can now be used in the BPEL process. In the process, correlation sets are used to associate messages with a business process instance. An initial message *initializes* the correlation set, i.e. defines the values of each property. The partner that sends this message is called the *initiator* of the set. All following messages must carry the identical values in the properties of the set to be routed to the given instance. All partners that use only follow-up messages are called *followers* of the set.

Before it can be used, a correlation set must be defined with a name, and a list of properties that constitute the set. Definition takes place inside the `correlationSets` section, which can be specified as part of the process itself, or a scope. Figure 38 defines a correlation set with the name `FlightOrder`. The set is based on the property defined in Figure 37.

```
<correlationSets>

     <correlationSet
          name="FlightOrder"
          properties="aln:FlightNo"/>

</correlationSets>
```

Figure 38: Definition of a Correlation Set

To use a correlation set, it must be attached to `invoke`, `receive`, and `reply` activities. The first message sent or received by one of these activities initiates the set; this must be specified as part of the activity.

`Receive` and `reply` activities only deal with one message. An invoke activity, on the other hand, may involve two messages (when making a synchronous call). In this case, an additional attribute is used to specify which messages the correlation set applies to: Only the outgoing message (`out` pattern), only the incoming message (`in` pattern), or both (`out-in` pattern).

The following example shows an interaction in which the BPEL process sends out a flight information message to a client, thereby initiating a correlation set, binding the set to the current instance. As the set is based on the flight number in the outgoing message via the `flightNo` property, the instance is now bound to that flight number. The BPEL process is the initiator of the set, and the client a follower.

The client may take its time deciding whether to book the flight. When ready, it calls the BPEL process back with an asynchronous call. This call is identified to be targeted at the BPEL process instance, because it carries the `flightNo` as well, and the receive activity is bound to the correlation set, which in turn is bound to the flight number via the `flightNo` property.

```
<!-- Receive flight request from client -->
...

<!-- Send out flight data -->
<reply
     operation="FlightRequest"
     ...
     <correlations>
          <correlation set="FlightOrder" initiate="yes" />
     </correlations>
</reply>

<receive
     operation="ConfirmTicket"
     ...
     <correlations>
          <correlation set="FlightOrder" />
     </correlations>
</receive>

<!-- Book the flight -->
...
```

Figure 39: Example Use of Correlation

2.3.9 BPEL execution environments

A BPEL process consists of the actual business logic (written in BPEL), a service description (WSDL), and possibly additional data types (XML Schema). To actually use a BPEL process, all these artifacts must be assembled and deployed into a BPEL runtime engine.

The BPEL language specification (Andrews et al. 2003) does not specify a deployment approach, nor a syntax for a deployment descriptor. Implementers of BPEL runtime engines must therefore come up with their own solutions. BPEL deployment must solve two problems:

- The BPEL process itself must be available to clients, specifying a WSDL service, port, and binding. One possibility to achieve this is to require BPEL developers to provide a full WSDL document, i.e., including the concrete section, and using the style and encoding specified in that document. However, as BPEL is designed to only use the abstract parts of WSDL, the concrete part is not relevant to BPEL itself, and could also be created by the BPEL engine.
- When instantiated, BPEL processes usually send messages to partners. The BPEL engine must know where these partners are physically located along with their invocation parameters in order to encode and send messages to their locations. During deployment, the deployer must be able to specify which WSDL service and port and which address is to be used for a partner link.

Note that although the abstract part of a WSDL file is sufficient for the implementing BPEL process, it is not for clients, who still need a WSDL service and port to be able to talk to the BPEL process. If the BPEL runtime generates the complete WSDL document, this complete document must be distributed to clients; not the original abstract document.

Many of the BPEL execution environments available are bundled with an editor, thus combining design and execution environment. Most BPEL editors also support the graphical creation of BPEL processes by means of a (proprietary) graphical notation. As mentioned before, BPEL was initially created by IBM, BEA, and Microsoft, which have the following BPEL products available:

- **IBM WebSphere Integration Developer**, an Eclipse-based integrated solution for editing, deploying, and running BPEL processes.
- **BEA WebLogic Integration**, a development- and runtime framework which also supports editing, deploying, and running BPEL processes.
- **Microsoft BizTalk**, a business process management server which includes support for BPEL. Designing BPEL processes is done in Visual Studio .NET or Visio.

However, other vendors have BPEL products available as well. For more information, see Juric 2006 (page 25), which lists over a dozen solutions for the execution and design of BPEL processes.

As examples for possible deployment and execution approaches, the next sections present the approaches taken by Oracle in the *Oracle BPEL Process Management Server* (Oracle 2006) and by *Active Endpoints* in the ActiveBPEL Engine (ActiveEndpoints 2006).

2.3.9.1 Oracle BPEL Process Manager

The free Oracle BPEL Process Manager is a solution for editing, deploying, and running BPEL processes. It includes a BPEL-enabled version of Oracle's JDeveloper IDE, and the BPEL PM server as the BPEL runtime.

To deploy a BPEL process to the Oracle BPEL PM Server, an XML-based deployment descriptor must be created, which specifies the following details about the BPEL process:

- BPEL source file name
- BPEL process name (used as an ID for the process)
- Locations of partner links
- Other configuration properties

The third item is of course most important, as this is the place where partner links are mapped to the actual physical locations of partners. For each partner link, it is possible to specify several options, among them:

- WSDL *location* of the partner link. This value is required and must point to the actual WSDL file of the partner. It is sufficient to provide the abstract part of the WSDL, which is all that's needed for *validation* of the BPEL process.
- WSDL *runtime location* of the partner link. This value points to the WSDL file used during runtime and must contain a complete WSDL description, including a WSDL service, port, and binding.

It is important to note that these settings apply to the partner links specified within the BPEL process, not to the partner link types. In a partner link, the two roles of the link type have already been specified. For a partner link which only has a role associated with the process itself (`myRole`), operations are invoked on the BPEL process and thus WSDL service, port and binding need not be present in the linked WSDL (they are then generated by the server).

However, for partner links which specify an actual partner, at least one of the locations must point to a complete WSDL description, or the engine will fail during partner invocation.

Figure 40 shows an example of an Oracle BPEL deployment descriptor. Two partner links are specified, one with the name `client`, and one with the name `LoanService`. The `client` is the actual client of the BPEL process; the operations needed by the client are specified in the BPEL WSDL file itself, which is linked here. The `LoanService` is a partner of the Web service which is deployed on the same server.

```
<BPELSuitcase>
   <BPELProcess
       src="LoanBroker.bpel"
       id="LoanBroker">

       <partnerLinkBindings>
          <partnerLinkBinding name="client">

             <property name="wsdlLocation">
                LoanBroker.wsdl
             </property>

          </partnerLinkBinding>
          <partnerLinkBinding name="LoanService">

               <property name="wsdlLocation">
                  http://localhost:9700/orabpel/default
                              /AmericanLoan/AmericanLoan?wsdl
               </property>

          </partnerLinkBinding>
       </partnerLinkBindings>
   </BPELProcess>
</BPELSuitecase>
```

Figure 40: An Oracle Deployment Descriptor

During the deployment phase, Oracle generates a service, port, and binding for the BPEL process if it is not already present in the process WSDL file. This binding uses SOAP and a literal encoding. The style depends on the specification of the part types in WSDL messages: If the `element` attribute is used, the generated service uses document style. If the `type` attribute is used, the generated service uses rpc style.

The actual deployment format is a JAR file containing all required source files (BPEL, WSDL, XSD, etc.) and meta information, most importantly the deployment descriptor itself. The Oracle JDeveloper IDE includes tools to generate this deployment JAR file.

Once deployed, the BPEL processes may be reviewed in a web-based console, which also offers complete in-depth information about completed or faulted BPEL process instances.

2.3.9.2 Active Endpoints ActiveBPEL engine and ActiveBPEL designer

The ActiveBPEL engine is an open-source BPEL execution environment created by Active Endpoints. The company also offers a free (non-open source) design environment (the ActiveBPEL designer), which is based on Eclipse and designed to interact with the engine.

To deploy a process to the ActiveBPEL engine, an XML-based deployment descriptor must be created, which specifies the same general details about the BPEL process as the Oracle descriptor. However, the specification of partner links is different.

- Firstly, ActiveBPEL allows the definition of a style and encoding for the concrete part of the BPEL process WSDL file (`myRole` partner links) instead of "inferring" them from the WSDL abstract part. One of four codes must be specified for style and encoding (all are SOAP-based):
    - `MSG`. Identifies document/literal style.
    - `RPC`: Identifies rpc/encoded style.
    - `RPC-LIT`. Identifies rpc/literal style.
    - `EXTERNAL`. Use an ActiveBPEL-specific style (proprietary).
- Secondly, an actual partner (`partnerRole`) is not only specified with a WSDL file, but by means of an endpoint reference as well. There are four types of endpoints available:
    - `static`. The deployment descriptor includes a WS-Addressing endpoint which identifies the partner.
    - `dynamic`. The deployment descriptor does not include an endpoint; the endpoint must be created dynamically.
    - `invoker`. The endpoint is passed in by using WS-Addressing.
    - `principal`. A special file is provided which identifies the endpoint.

It should be noted that the specification of a style and binding for the created WSDL is required, i.e. the concrete part will *always* be created, *overwriting* the service, port, and binding information that may have already been specified.

```
<process
     location="loan approval.bpel"
     name="bpelns:loanApprovalProcess"
     ...
     >
  <partnerLinks>
     <partnerLink name="customer">
        <myRole allowedRoles="" binding="RPC" service="ApproveLoan"/>
     </partnerLink>
     <partnerLink name="approver">
        <partnerRole endpointReference="static">
           <wsa:EndpointReference>
              <wsa:Address>(ignored)</wsa:Address>
              <wsa:ServiceName
                  PortName="SOAPPort">
                        approver:LoanApprover
              </wsa:ServiceName>
           </wsa:EndpointReference>
        </partnerRole>
     </partnerLink>
  </partnerLinks>
  ...
</process>
```

Figure 41: An ActiveBPEL Deployment Descriptor

When using the static type for specifying an endpoint, a WS-Addressing endpoint must be specified in the deployment descriptor. As noted before, a WS-Addressing endpoint *requires* the specification of an address and *allows* specification of a WSDL service and port. However, by default, ActiveBPEL ignores the address. This allows using the address already provided in the concrete section of a given WSDL document.

Figure 41 gives an example of an ActiveBPEL deployment descriptor. It has two partner links: The first one is a link with only a `myRole` specification; it is set to RPC style invocation. The second one is an invoked partner link, which is specified with a static endpoint reference type, pointing to a concrete service and port from the partner WSDL concrete section. The URL in these endpoint types is ignored by ActiveBPEL (this behavior may be changed by certain options).

The actual deployment format is, again, a JAR file, containing all needed source files and meta information (including the deployment descriptor). The JAR file has the file ending ".bpr" for *Business Process aRchive*. The ActiveBPEL designer contains tool support for creating such archives.

The ActiveBPEL engine also offers a web-based BPEL front end, in which data from completed or faulted BPEL processes may be reviewed.

2.3.10 Summary

In this section, the Business Process Execution Language (BPEL) has been presented, including its syntax, basic and advanced concepts, and the missing deployment link between the language artifacts and execution of BPEL processes; the latter by means of two commercial examples.

The BPEL language is rather new; the current version is 1.1, which has been presented in 2003. BPEL 1.1 is not a standard, but a specification created by some leading industry vendors. OASIS is currently working BPEL 2.0, which will be published as the first "real" BPEL standard.

Because of this situation, it is still too early to pass judgment on BPEL; it is also difficult to find any sources about an actual use of BPEL in both the commercial and academic sector. However, the experiences gained in this thesis can be summarized as follows:

- BPEL processes are highly complex entities. Despite graphical editors, BPEL processes cannot be designed by point-and-click, and designing them requires a structured approach, starting from solid WSDL service descriptions and moving on to the BPEL process, scopes, structured activities, and finally basic activities.
- Working with BPEL requires a solid understanding of all underlying standards, starting from XML, XML namespaces, XML Schema, XPath, and including SOAP, WSDL, and WS-Addressing, as underlying concepts "leak" through to the BPEL level in many places.
- The complexity of BPEL processes and BPEL deployment is noticeable in existing tools, which appear rather fragile. Especially the integration of different artifacts, like XML Schema definitions, WSDL descriptions, and BPEL process definitions can be improved. BPEL editors are a long way from conventional language IDEs like Eclipse or IDEA.
- The actual deployment of BPEL processes, including the creation of concrete WSDL parts and binding partner processes to partner links is highly vendor-dependent and also still very complex.

However, even it BPEL is so far not used in production environments, it is certainly being evaluated, and may fulfill its promise once the technology matures.

2.4 Software testing

Software testing is a popular quality-improvement technique supported by both academic research and commercial experience. In "The Art of Software Testing", Myers offers a definition of software testing:

Testing is the process of executing a program with the intent of finding errors (Myers 1979).

This definition is simple but precise. It hints at the attitude a software tester should have, which is not to prove a software works correctly (i.e., has no errors), but rather to prove that it doesn't work (i.e., has errors). The former is impossible anyway – as pointed out by Dijkstra (Dijkstra 1970), testing can only prove the existence of errors, but not their absence.

Software testing can take many forms (McConnel 2004). The most well-known are:

- **Unit testing**. A unit test is used to test a single class, routine, or more generally, *component*, in isolation from the rest of the system it belongs to.
- **Integration testing**. An integration test is used to test combined, or *integrated*, components of a software system.
- **Regression testing**. Regression testing employs repeated execution of test cases to find bugs in updated software which has already passed the tests.
- **System testing**. System testing is the process of testing the complete, final system, which includes users and interactions with other systems.

Testing can be further divided into two general groups:

- **Black-box testing**. In black-box testing, the program under test is viewed as a black box – the tester cannot see, or is not aware of, the inner workings of the program.
- **White-box testing**. In white-box testing, on the other hand, the tester is aware of the inner workings of the program, thus viewing the program as a white (or rather, *transparent* or *glass*) box.

In this thesis, the focus is on unit testing. The following sections will therefore given an overview of unit testing concepts and technology.

2.4.1 Unit testing

As pointed out above, unit testing is the process of testing a single component, named a *unit*, of a program in isolation. Unit testing is performed for the following reasons (Myers 1979):

- Unit testing provides a base for later integration testing, as it requires components to be identified and managed separately.
- Debugging is easier in unit testing, as every bug found by a unit test is known to originate in the current unit.
- Unit testing of several components can be performed in parallel.

Unit testing is largely white-box oriented, as it focuses on the correct implementation of a low-level component, which will only later be integrated into a complete system. The intent is to *find errors in the program's logic* (Myers 1979*)*. This is rather different from integration and system testing, where user acceptance and interaction with other systems play a more important role.

The isolated testing approach of unit testing requires special attention. Normally, a component of a software system has links to other components in order to fulfill its task. However, as the intent of unit testing is to test a component by itself, such links often require setup routines which establish a context in which the unit test may run. If many components are involved, this may get rather tedious, or even unrealistic. As a remedy, the components behind such links are replaced by custom implementations, which *imitate* the component behavior as far as it is needed for the test. Such replacements have come to be known as *mocks* (Mackinnon et al. 2001).

Besides facilitating unit testing, mocks also offer additional benefits, as they allow introducing faulty or borderline behavior on behalf of a partner component, thus also testing the unit for stability and fault tolerance. In many ways, mocks allow a more thorough test of a component, as the test input now comes from all sides, and not only from the intended client of the component.

Unit testing is perceived by many as a methodology which not only greatly helps in finding bugs, but which can also establish *trust* in the code on behalf of the programmer, which in turn enables him to refactor his code where and when this is needed. Besides this rather psychological benefit, unit testing has also been proven to improve quality in practice (Ellims et al. 2004).

When considering all test approaches, unit testing is also the most automated testing method. The inner logic of a component can (mostly) be tested by executing *test code* as opposed to manually *using* the software. Nowadays, unit testing almost always includes regression testing. There are many tools available for unit testing, many of which focus on automated, repeatable testing. Unit testing has been wholly embraced by the Extreme Programming community (Beck 2000) and in the area of Test-Driven Development (Beck 2003).

Unit testing is supported by unit test frameworks, which are available for all kinds of programming languages. The most important family of unit test frameworks is the *xUnit* family (Hamill 2004), with SUnit as the initial contribution. The xUnit family is discussed in the next section.

2.4.2 The xUnit family

The xUnit family has its beginnings in SmalltalkUnit (SUnit), a unit test framework for the Smalltalk programming language introduced by Kent Beck in 1999 (Beck 1999). Later, Erich Gamma ported SUnit to Java, thus creating JUnit (Gamma et al. 2006), the most widely used and extended unit test framework available.

Today, dozens of unit testing frameworks build on the same principles as SUnit and JUnit, thus creating the xUnit family. Some of the more popular frameworks include CppUnit (for C++), NUnit (for .NET), PyUnit (for Python), vbUnit (for Visual Basic), and utPLSQL (for Oracle's PL/SQL).

The xUnit family frameworks offer the following functionality to testers:

- **Support for writing tests**, for example in the form of abstract base classes to be extended, utilities for test setup, or build scripts.
- **Functionality for running tests** by providing command line or graphical test runners which execute the tests along with the software under test.
- **Result presentation**, i.e. gathering of test results and presenting them to the user.

The concrete implementation of a xUnit test framework is dependent on the target language and can thus be very different. However, all xUnit test frameworks are built around the same basic concepts, which include the notions of a test case, test suite, test fixture, test runner, and test result.

- **Test Case**. Software testing and the xUnit family in particular are based on the fundamental concept of a test case. In xUnit, a test case is the place where the test logic resides – i.e., the actual statements which invoke code in the unit under test and verify the results.
- **Test Suite**. A test suite is an aggregation of multiple test cases. Through a test suite, the tests cases can be run as a whole.
- **Test Fixture**. A test fixture can be seen as an environment for multiple tests. Such an environment is *set up* before a number of tests are executed, and *teared down* after the tests complete. A test fixture is normally included as part of a test suite or test case.
- **Test Result**. The test result is an abstraction of the result of a test run. In xUnit terms, a test may pass, fail, or have an error; the difference between a failure and error being that a failure is an application-level problem, whereas an error is a severe technical malfunction. A test result contains information about the final status of the tests which have been run.
- **Test Runner**. Each xUnit framework contains one or more test runners, which are responsible for executing the test cases or test suites, thereby producing test results.

The success of the xUnit family and, especially, of JUnit has to do with the new approach on testing which these unit test frameworks provide. For a long time, testing has been regarded as a separate phase in the software development process, carried out after the main coding activities. This approach has recently come under fire, most notably from the Extreme Programming community mentioned above. A new approach was formed, which sees testing as a parallel activity to coding, or even (in the case of Test Driven Development) as one step ahead of coding.

The main intent behind the xUnit frameworks is the *facilitation* of unit testing. It enables programmers to write tests without a noticeable "mode switch" from coding to testing, and makes it easy to run tests time and time again. As Kent Beck and Erich Gamma have put it, "*programmers love writing tests*" with JUnit (Beck et al. 2006) – and it is this fact which makes the xUnit family so powerful.

3 Concepts of BPEL composition testing

In this chapter, the theoretical underpinnings of this thesis are presented: Possible approaches to BPEL composition testing, an abstract architecture for BPEL composition testing frameworks, and design decisions for a concrete testing framework.

The BPEL specification describes a complete high-level programming language, which includes all options for creating very complex programs. Many books have been written about computer programming, and many of them place a great emphasis on writing bug-free code. Nevertheless, after decades of evolution in programming languages and tools, programs still contain faults, regardless of programming language or tool support.

Programs written in BPEL will be no exception. In fact, it can be expected that the number of bugs in BPEL programs will be rather high, due to the following three reasons:

- The BPEL language resides on top of an enormously complex Web service technology stack, which has been designed for extensibility, handles complicated business scenarios, and is rather new.
- The BPEL language is even newer, thus common pitfalls have not yet been identified.
- The BPEL language closely integrates with two other specifications: Variables are XML Schema-based, and all conditions and assignments in the language are specified by using XPath. The interdependence of BPEL programs with artifacts from these two languages is particularly complicated.

As has already been pointed out in the previous chapter, software testing is an undisputed requirement for all software development processes. Extending testing tools to support the service composition approach and BPEL in particular is a logical step for the software engineering community, ensuring quality in BPEL programs as in other languages. The focus of this thesis is thus to introduce testing tools for BPEL, enabling developers to take industry best practice with them when moving to service composition.

As noted in chapter 2, BPEL can be used for creating two kinds of business processes: Abstract and executable processes. While abstract processes follow the choreography paradigm and are intended to convey a certain collaboration concept, executable processes follow the orchestration paradigm and are intended to be executed. Software testing is a technique which requires the execution of a program; therefore, the focus of this thesis is on executable business processes.

This chapter is an extended version of material presented in Mayer et al. 2006. Section 3.1 introduces basic concepts of BPEL composition testing. Section 3.2 presents a generic BPEL unit testing architecture, followed by the concrete testing framework *BPELUnit* in section 3.3.

3.1 The realm of BPEL composition testing

When comparing BPEL to conventional programming languages, it becomes clear that the focus of BPEL is rather different – it is at the same time more targeted at the particular job of service integration and more open with regard to data handling. Also, its execution environment differs radically from many other languages. BPEL composition testing therefore has different requirements as well. The following three sections introduce the form of testing used, identify special requirements of BPEL composition testing, and present a general BPEL composition testing approach.

3.1.1 Introducing BPEL composition testing

As laid out in the previous chapter, there are many forms of testing, and one of these forms must be selected in order to identify the requirements for a testing tool.

BPEL programs are components in a software system, and depending on the current phase of the software development project, can be tested by different approaches. As pointed out in the previous chapter, the following three approaches to testing are amongst the most well-known:

- **System testing**. *As a global* testing approach, system testing has nothing to do with the BPEL language as such; existing testing tools can be employed here.

- **Integration testing.** With regard to BPEL compositions, integration testing is in fact Web service testing – a BPEL process is integrated with other Web services (its partners) and is tested in combination with them. Web service testing tools are readily available and can be used for this purpose (for more information on existing solutions, see section 3.4).
- **Unit testing.** Unit testing directly deals with a BPEL composition, testing it in isolation (specifically, separated from partners).

Of all testing forms, unit testing is the closest to the actual code of a system. As pointed out in the previous chapter, unit testing mostly uses a white-box approach, testing the internal logic of a component. In this thesis, the focus lies on such *white-box unit testing*. The goal is to create a framework which can be used to *execute* a BPEL process under certain conditions, thus testing its internal logic.

Software development processes include many roles, for example the roles of developer and tester. Testing in general and unit testing in particular may be executed by different people working in different roles.

One option in software testing is testing-by-developer, which means that the developer of a certain piece of code will also be the one to write and execute the test cases for this code. There has been much discussion about whether the developer can ever be in the right frame of mind for this, as testing is a rather destructive approach. On the other hand, a developer knows about weak points in his code and can get rather productive by testing those first. Testing-by-developer is used in agile development approaches (for example, in the Extreme Programming and Test First communities, see Beck 2000).

The second option in software testing is testing-by-tester. The tester is usually part of a quality assurance team, which gets to test the software after the developers are done. The basic assumption behind this approach is that an outsider will take a more rigorous approach to testing (Myers 1979). Testing-by-tester is mostly used in non-agile (classic) development approaches, or as an addition to developer testing in agile projects.

In this thesis, the focus is on testing BPEL as a programming language, and on creating a testing framework similar to the frameworks of the xUnit family, enabling automated, repeatable white-box testing of BPEL processes. It therefore employs *developer testing*. The framework presented in later sections will contain low-level tools intended for use by developers and is not intended to be used by testers without a solid background in BPEL and Web service technology.

3.1.2 Requirements of BPEL composition testing

The purpose of white-box unit testing is to find bugs in the code itself. This applies to BPEL as well. As BPEL is rather new, it is very difficult to find reports of successful – or unsuccessful – deployments of BPEL processes in industry projects, let alone long-time studies of bug occurrences in BPEL programs.

However, from the experience gained in this thesis, the following three areas in BPEL programs are assumed to be more error-prone than others:

- **Data handling**. BPEL employs XPath statements to copy data from and to variables and for condition checking. These statements extract data from or copy data to XML Schema-based XML elements. Such elements are complex structures, and it is quite easy to find an XPath solution which caters for some, but not all possible elements compatible with a Schema. A BPEL program may thus run without error for most of the input and output data, but fail in corner cases.
- **Parallel activity handling**. When multiple partners are called simultaneously, care must be taken that the correct path is executed regardless on which partner returns first. BPEL offers the possibility of employing condition-based activity graphs, which require thorough planning to handle all possible arrival sequences.
- **Fault and compensation handling**. Handling faults in BPEL is non-trivial, as many different types of faults may occur and must be handled in a comprehensible way. Introducing a custom fault handling concept is a necessary step for BPEL programmers, as the language itself does not contain mechanisms for propagating faults to clients. Along the same lines, compensation activities must be created to reverse the effects of a scope.

A framework for BPEL composition testing should take these three areas into account by defining adequate specialized testing constructs. BPEL composition testing also has some special, BPEL-related requirements, as it differs in many ways from conventional language testing. Three important differences are to be noted:

- **Surrounding middleware**. BPEL programs run inside a BPEL middleware, i.e. the code is not directly accessible as it would be in conventional programming languages – the test must interoperate with the middleware to be able to execute a BPEL program. The middleware itself is not under test; however, any problems which occur in the middleware should at least be presented to the user.
- **Different data level**. BPEL programs have a different data level than other languages. As has been pointed out in the previous chapter, BPEL processes use WSDL messages, XML Schema elements, or XML Schema simple types as variable types. These types are defined in WSDL and neither correspond to the invocation data (the data on the wire, i.e. SOAP) nor the data handled in partners (which are usually implemented in conventional programming languages).
- **Parallel activities**. BPEL has built-in support for parallel activity handling which requires special attention. In particular, the number of possible paths multiplies when using such activities and a test framework must contain special functionality to offer the ability to test such paths. While conventional programming languages do offer support for parallel behavior (for example, threads in Java), use of these is not as common as in BPEL, and in fact, special testing tools for threads must be used in addition to JUnit.

Besides these BPEL-specific requirements, all requirements for conventional, xUnit-based test frameworks apply as well, among them:

- Enabling fast "mode switching" between programming and testing.
- Being easy to use.
- Following the well-known testing concepts of the xUnit family, i.e. using the same names and descriptions where applicable.
- Enabling repeatable testing, with a UI or headless, producing reports to be used for governance or other controlling purposes.

To sum up, a BPEL unit testing framework should respect the xUnit family standards while putting emphasis on the special problem areas of BPEL as well as BPEL-specific requirements.

3.1.3 Test approach

The way BPEL processes communicate with their surroundings – via standard Web service calls – is of great advantage for testing, as opposed to conventional languages which do not have this kind of unit separation. In fact, defining units when testing conventional programs often requires a specific “testable” program architecture using interfaces (although some are of the opinion that such code actually furthers program quality and understanding (Beck 2003)) to allow separate testing of so-defined units.

In BPEL, this separation is already in place: A unit is a BPEL process, and its interfaces are clearly defined through WSDL:

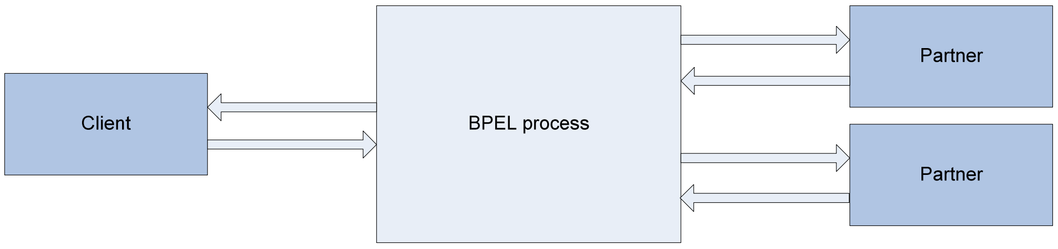


Figure 42: A BPEL Process and its Partners

As said before, the intent of this thesis is to provide BPEL developers with a means to create white-box unit tests. As unit testing is the process of testing a component in isolation, some way must be found to test a BPEL process without all of its partners attached. This can be achieved by using *mocks*, which – somehow – replace the partners of the process by custom implementations specifically created for the test.

Thus, the natural approach to testing BPEL processes is to create a *harness* around the process under test (PUT), enabling a test case to receive data from the BPEL process and to feed data back in at each of the interfaces the process provides – i.e., every operation offered by the service and each operation invoked by the service.

It is important to note that the introduction of mocks is an absolute requirement for BPEL composition testing, as opposed to normal Web service testing, where a custom client is sufficient. The test harness replaces both the client and all partners of a BPEL process:

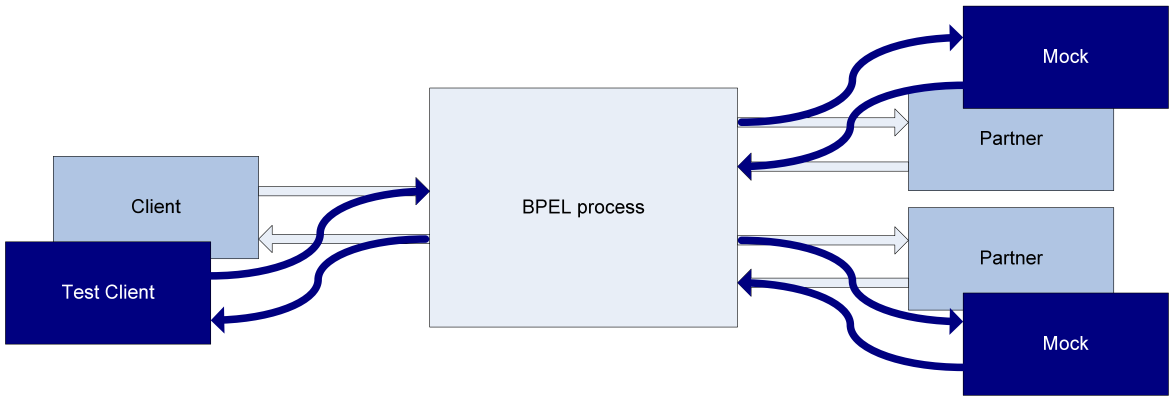


Figure 43: A Test Harness

There are several possible implementations for achieving this, which are described in detail in the next section.

3.2 A BPEL unit testing architecture

In this section, a generic, layer-based approach for creating BPEL composition testing frameworks is presented, which is later used for the design of a concrete framework. As a side effect, this layer-based model can be used for classifying existing frameworks or implementations of other frameworks.

The architecture consists of several layers which build upon one another, as outlined in Figure 44. The functionality of each layer can be implemented in various ways, which are pointed out in the subsequent sections.

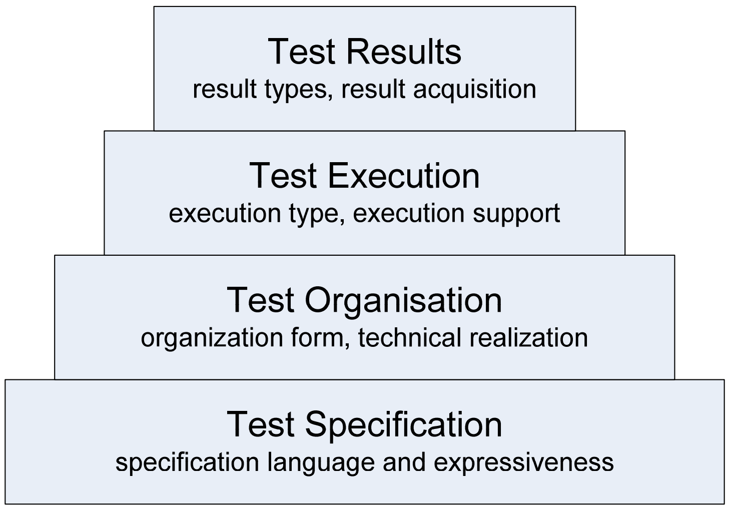

Test Results
result types, result acquisition
Test Execution
execution type, execution support
Test Organisation
organization form, technical realization
Test Specification
specification language and expressiveness

Figure 44: BPEL Composition Testing Architecture

The first (bottom) layer is concerned with the test (case) specification – i.e., how the test data and behavior are formulated. Building on this, the tests must be organized into test suites (test organization layer).

To achieve results, a test – and therefore also the process under test – must be executed (test execution layer). The results must then be gathered and logged or presented to the user (test results layer).

3.2.1 Test specification

Testing a process means sending data to and receiving data from its endpoints, according to the business protocol imposed by the process under test and its partner processes.

BPEL interfaces are described using WSDL port types and operations. As pointed out in chapter 2, the WSDL syntax lacks a description of the actual *protocol* of a Web service, i.e. which operation must be invoked after or before which other operation (for a discussion, see Alonso et al. 2004, pp. 137). This is particularly relevant for asynchronous operations. A testing framework must provide a way for the tester to specify such a protocol and to follow it in the test harness. As a BPEL implementation makes no real distinction between the clients of the process and other invoked processes, this applies to the client as well as all partner processes.

As for the information flow between the BPEL process and its partner processes, data can be differentiated between incoming and outgoing data from the perspective of the test harness developer:

- **Incoming data** (from the point of view of the test) is data sent by the PUT. Each expected data package must be analyzed by the test for correctness.
- **Outgoing data** (from the point of view of the test) is test data sent back into the PUT to achieve a certain goal, i.e. some branch should be taken as a result, a fault handled or thrown, or a compensation handler activated. The test specification must provide a way for describing such test data.

The test specification must thus validate the correctness of incoming data as well as create outgoing data. As pointed out by Li et al. 2005, incoming data errors can be classified into three types:

- incorrect content,
- no message at all, when one is expected, and
- an incorrect number of messages (too few or too many).

There are several ways of formulating the test specification to achieve these goals. The following two examples are the most extremes:

- **Data-centered approach**: for example using fixed SOAP data, augmented with simple rules. Incoming data from the process is compared against a predefined SOAP message (which for example resides in some file on disk). Outgoing data is predefined too, read from a file and sent to the process. A simple set of rules determines if messages are expected at all and takes care of sending replies. Needless to say, this approach is very simple, but also least expressive.
- **Logic-centered approach**: for example using a fully-fledged programming language for expressing the test logic. A program is invoked on each incoming transmission which may take arbitrary steps to test the incoming data. The outgoing data is likewise created by a program. This approach is very flexible and expressive, but requires a lot more work by the test developer.

Of course, there are several approaches in-between. A data-centered approach could use a simple XML specification language to allow testers to specify test data at the level of BPEL, i.e. XML-typed data records instead of SOAP messages. A logic-centered approach could use a simple language for expressing basic conditional statements ("if the input data is such-and-such, send package from file A, otherwise from file B").

The choices made here have significant influence on the complexity of the test framework and the ease of use for the developer. In most cases, the complexity of the framework reduces work for the developer, and vice versa.

Beside the questions of expressiveness of the logic and simplicity for the tester, two additional requirements must be considered:

- **Automation**: The ultimate goal of a BPEL composition testing framework is automated repeatable testing, which means the test must be executable as a whole. This indicates that however the test is specified, the specification must be unambiguous, machine-readable and executable. The more sophisticated the test logic, the more complex the test execution will be.
- **Tool support**: It should be possible to automate at least some of the steps for creating the test specification, thereby relieving the tester of the more tedious tasks and letting him focus on the actual problem.

Regardless of how the test specification is implemented, it will be used by the developer for describing *BPEL test cases*. A BPEL test case contains all relevant data for testing a certain path in a BPEL process.

3.2.2 Test organization

As pointed out before, the test specification allows users to define test cases. While a test case contains all necessary information for testing a certain path of a BPEL process, it is not yet bound to a specific BPEL process, which may be identified by an URL, a set of files, or something completely different. The test organization must provide a way to link the test cases to a concrete BPEL process for testing.

Additionally, the true value of automated testing comes from executing many test cases as often as possible. Therefore, it is necessary to be able to group tests into composite tests (*test suites*).

For these two purposes, the test suite concept of conventional xUnit approaches is extended as follows:

- A BPEL test case will always be executed as part of a test suite.
- The test suite provides the *test fixture* for all enclosed test cases. This fixture contains the link to the BPEL process under test.

By using this approach, the fixture is globally specified in the suite and applicable to all test cases, which do not need to specify the BPEL process binding again.

Another important aspect of test organization is the integration of the test framework into the overall development process. For example, tracing requirements throughout the development cycle is important to track and react to changes. To permit requirements tracing throughout the testing process, it must be possible to augment the test cases with custom, requirements-related meta-data, and to query this data upon test completion.

There are two basic approaches to test organization:

- **Integrated test suite logic**: The first approach is to integrate test organization with test specification. This is only possible when a sophisticated test specification method is in place (for example, when using a high-level language). This approach has the benefit of being very flexible for the test developer. There is a huge drawback, however – the test framework has no way of knowing about composite tests and is not able to list the results separately.
- **Separate test suite specification**: The second approach is to allow formulation of separate test organization artifacts. These artifacts could include links to the actual test cases and the test fixture.

As in the previous section about test specification, it is also important here to stress the importance of automation and tool support for test organization, as the organization artifacts are the natural wrappers for the test specification.

3.2.3 Test execution

A BPEL process is an executable program which must be executed in order to test it. For normal execution, BPEL processes are usually deployed into a BPEL engine, then instantiated and run upon receipt of a message triggering instance creation. However, for testing a BPEL process, other mechanisms may be used as well.

BPEL composition testing means executing a BPEL process with a test harness around it, handling input and output data according to the test specification. This can be done in several ways. The following two approaches are the most obvious ones:

- **Simulated testing**: Simulated testing, as defined here, means the BPEL process is not actually deployed in the usual sense and invoked afterwards by means of Web service invocations. Instead, the engine is contacted directly via some sort of debug API and instructed to run the PUT. Through the debug API, the test framework closely controls the execution of the PUT. It is therefore possible to intercept calls to other Web services and handle them locally; it is also possible to inject data back into the PUT. This approach is taken by some editors currently available for manual testing and debugging.
- **Real-life testing**: Real-life testing, as defined here, means actually deploying the PUT into an engine and invoking it using Web service calls. Note that this means that all partner Web services must be replaced by mock Web services in a similar way, i.e. they must be available by Web service invocation and be able to make Web service calls themselves. The PUT must be deployed such that all partner Web service URIs are replaced by URIs to the test mocks.

Both approaches are heavily constrained by the existing (or rather, non-existing) infrastructure:

- Simulated BPEL execution only works if the engine supports debugging; i.e. has a rich API for controlling the execution of a BPEL instance. Whilst most engines do support such features, they are unfortunately in no way standardized. To avoid vendor lock-in, a test framework must therefore factor out this part and create adapters for each BPEL engine to be supported, which may get rather tedious.
- Real-life BPEL execution requires the process to be deployed first, binding the PUT to custom (test) URIs for the test partner processes. However, most engines rely on custom, vendor-specific deployment descriptors, which the test framework must provide, and which are not standardized as well. Furthermore, the BPEL specification allows dynamic discovery of partner Web services. Although frequent use of such features is doubted (Alonso et al. 2004), a framework relying on real-life test execution will have no way to counter such URI replacements.

There are certain correlations between the two specification approaches discussed in section 3.2.1 and the two execution types. For example, the test framework can directly use predefined SOAP messages in the case of simulated testing; real-life execution requires Web service mocks, which can be formulated in a higher-level programming language.

However, other combinations are also possible and depend on the amount of work done by the framework. It is relatively easy to create simple Web services out of test data, and simulating BPEL inside an engine does not mean the test framework cannot forward requests to other Web services or sophisticated programs calculating a return value.

The choice basically leads to the question of parameterization: On the one hand, all logic related to the test may reside in the framework, i.e. the framework itself simulates the partners; or on the other hand, all logic may reside in user-provided scripts or programs, and the framework just passes message data from the BPEL process to the user programs.

As in the xUnit family, the part of the framework responsible for executing the test is called the *test runner*. There may be several test runners for one framework, depending on the execution environment.

3.2.4 Test results

Execution of the tests yields results and statistics, which must be presented to the user at a later point in time. Many metrics have been defined for testing (Hong et al. 1997), and a testing framework must choose which ones – if any – to calculate and how to do this.

The most basic of all unit test results is the boolean test execution result which all test frameworks provide: A test succeeds, or it fails. Failures can additionally be split into two categories, as is done in the xUnit family: an actual failure (meaning the program took a wrong turn) or an error (meaning an abnormal program termination).

Furthermore, test metrics can be calculated. A very common test metric is the *code coverage* metric which has many flavors. It indicates the percentage of code which has been executed in a test case (or a test suite, for the matter). Coverage of 100% indicates each code statement (in case of statement coverage) has been executed at least once. This does not mean that each path has been taken; this is indicated by the path coverage metric (also ranging from zero to 100%).

The more sophisticated the metrics, the more information is usually required about the program run. This is an important aspect to discuss because control over the execution of a BPEL process is not standardized as pointed out in the last section. For example, it is rather easy to derive numbers on test case failures, but activity coverage analysis requires knowledge about which BPEL activities have actually been executed. There are several ways of gathering this information:

- **During BPEL simulation**, APIs may be used to query the activity which is currently active. However, these APIs, if they exist, are vendor-specific.
- **During BPEL execution**, the invoked mock partner processes are able to log their interactions with the PUT. It is thus possible to detect execution of some PUT activities (i.e. all activities which deal with outside Web services). However, this requires additional logic inside the mock partner processes which will complicate the test logic. Conclusions about path coverage may also be drawn from this information, but they will not be complete as not all paths must invoke external services.
- **As a follow-up.** It has been suggested (Li et al. 2005) to use log files produced by BPEL engines to extract information about the execution of particular instance, and to use this information to calculate test coverage. Such logs, if they exist, are of course again vendor-specific.

The calculated test results must also be presented to the user. A BPEL test framework should make no assumptions about its environment, i.e. whether it runs with a graphical UI, or headless on a server. For all these cases, the test runners should be able to provide adequately formatted *test results*; for example, a graphical UI for the user, or a detailed test result log in case of headless testing.

With this explanation of the test result layer, the description of the four-layer BPEL composition testing framework architecture is complete. In the next section, a concrete instance of this generic framework is presented.

3.3 BPELUnit

The previous section provided a generic architecture for BPEL composition testing frameworks and presented the choices to be made when conceiving such a framework. In this section, a concrete framework instance based on the discussed architecture is presented, which is called *BPELUnit*. Conceptually, BPELUnit is a member of the xUnit family, although part of the realization differs from conventional xUnit approaches.

This section describes the choices made for each of the framework layers, and problems to be solved when taking this approach.

3.3.1 Test specification

The most important choice to be made for each BPEL composition testing framework is how to specify the test logic. This is the first design decision to be made in the layered architecture described in the previous section.

BPELUnit focuses on developers who interleave testing and coding during development. To support this development style, a test framework should allow rapid testing, i.e. creating and running tests should be easy and fast.

Formulating BPEL test cases can be greatly facilitated for the tester by creating a specialized language which allows specification of which data is to be sent to the PUT, and which data is expected at each partner service – and then let the testing framework do the rest. Creating such a language requires decisions with regard to the following two items:

- **Data specification**. Data can be specified at various levels – the lowest possible level being SOAP, and the highest possible level depending on the implementation of the corresponding Web service (BPEL process, and partners).
- **Interaction details**. Interaction details, i.e. the protocol to expect at a partner, and the protocol to follow at the client side, can be specified in various ways.

As pointed out before, the *native* BPEL data format is literal XML, described as WSDL message types, XML Schema elements, or XML Schema simple types. BPEL developers directly deal with XML, creating XPath expressions for selecting elements from variables or creating new elements. It is therefore believed that this is also the best language for specifying data to be sent to the PUT.

The data specification of BPELUnit thus also employs such *literal XML data* for sending data to the PUT – from whichever direction (i.e. client, or partner).

One could use the same format to check incoming data – i.e., compare the XML node-by-node. However, data from the PUT may contain random elements like dates or auto-incremented numbers, which may or may not be relevant for the test. Instead of specifying which data is not relevant in a complete data package, the opposite approach is adopted: Specifying which data is relevant by means of XPath expressions. A check of incoming data thus consists of two items:

- An XPath expression used for selecting a value from the incoming data.
- A value which is then compared with the result of the XPath expression.

This approach is very flexible: An XPath expression can be used for retrieving a single value from the XML data (for example, a string, a boolean value, or an integer), and by using XPath functions, it can also be used for counting elements in the XML data or even performing sophisticated calculations.

With the data format specified, the discussion can move on to *interaction details*. Testing the PUT means verifying the correctness of the implemented business protocol. To do this, the business protocol of the client and the partners must be simulated in order to provoke and test the reactions of the BPEL process. A business protocol consists of *interactions* (or *data exchange patterns*): Each interaction either sends data to the PUT, receives data from the PUT, or both:

- **Sending data**: Data, specified as literal XML as mentioned above, is sent to the PUT.
- **Receiving data**: Data is expected from the PUT; the data is verified with XPath expressions as mentioned above.

Interactions can be categorized into three general forms, yielding six basic interaction types:

- **One-way interactions** (receive-only and send-only). Although probably rarely used, these interactions can be used for single-message calls.
- **Two-way synchronous interactions** (send-receive and receive-send). These are the most obvious interactions. The first is intended to be used in the client of the PUT, whereas the second will be used by partners.
- **Two-way asynchronous interactions** (send-receive and receive-send). Although in fact consisting of two one-way operations, it is best to think of these interactions as a logical unit. They will be used in a similar fashion to their synchronous counterparts.

To form a test case, testers need to specify *sequences of these interactions* for the client as well as every partner of the PUT. By chaining such interactions together, it is possible to shape different interaction protocols with the PUT on a case-by-case basis.

3.3.1.1 Interaction sequences

The sequences of interactions mentioned above lie at the heart of BPELUnit. A single atomic interaction with the PUT is called an *activity*. Sequences of activities are called *tracks*: The framework runs one track for each partner (and one for the client). Tracks run in parallel and are independent of each other.

Figure 45 shows a sequence diagram of a typical interaction between the client and partner tracks of the framework and the PUT. The PUT is a simple BPEL process which talks to two partners to achieve its goal: One partner is invoked asynchronously, and the other synchronously. The PUT itself offers a synchronous operation, which is tested by the client track.

The partner tracks and the client track each test incoming data from the PUT, and send predefined literal data to the PUT.

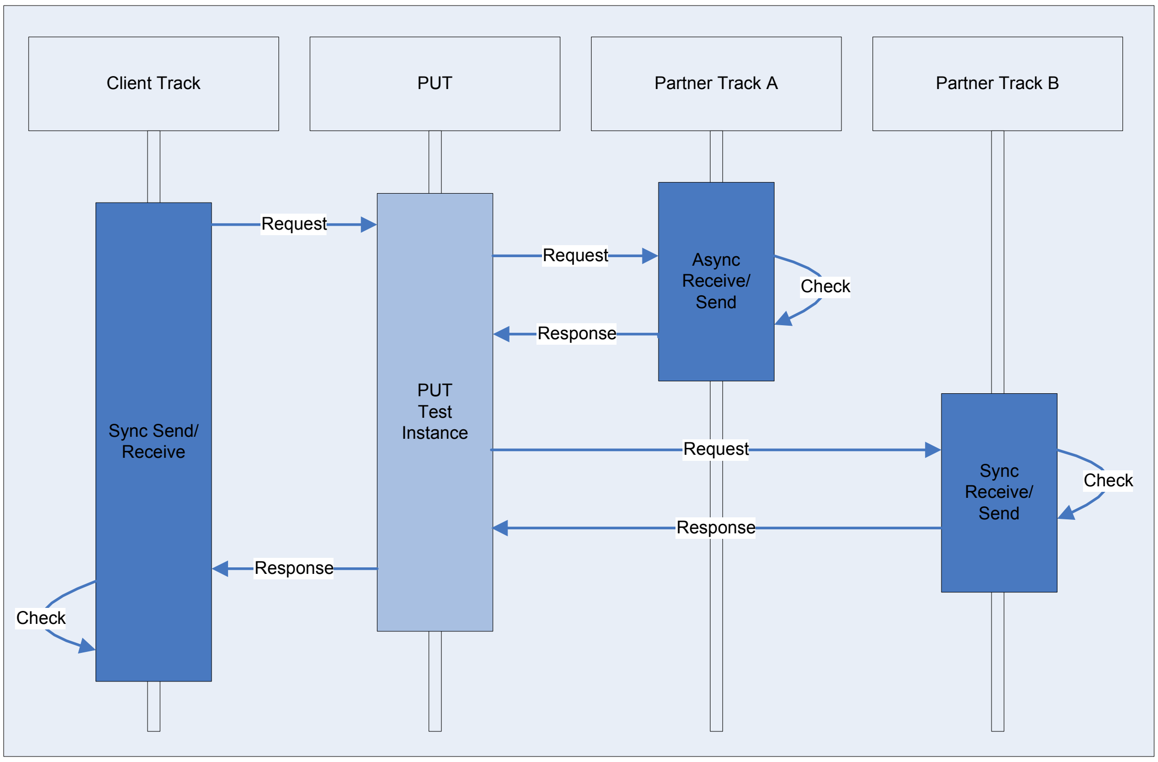


Figure 45: A BPEL Test Case Sequence

The BPELUnit activities relate to WSDL port types and operations. Each activity corresponds to at least one WSDL operation:

- **One-way and synchronous two-way activities**. These activities correspond to exactly one WSDL operation inside a given WSDL port type. A one-way activity corresponds to a one-way operation, which means that the WSDL operation only contains one message. A synchronous two-way activity corresponds to a request-response operation, which means that the WSDL operation contains at least two messages (one for input, one for output, and additional faults).
- **Asynchronous two-way activities**. Asynchronous two-way calls are implemented in BPEL by using two WSDL operations: One operation is invoked at the beginning of the call; and another is used for the callback. Asynchronous two-way activities thus correspond to two one-way WSDL operations, which will usually belong to different port types and thus ports and services.

Note that depending on the track of the activity, the operation will be specified in different WSDL namespaces:

- If an activity belongs to the client track, the interaction is targeted at the BPEL process. The corresponding WSDL operation thus resides in the *target namespace of the BPEL WSDL definition*.
- If an activity belongs to a partner track, the interaction is targeted at the test framework simulating the partner. The simulated WSDL operation thus resides in the *target namespace of the partner WSDL definition* and will be executed *on behalf* of that partner.

It is important to note that the WSDL operations will always be one-way or receive-response WSDL operations – no solicit-response or notification operations are involved (in fact, BPEL does not support these operations).

The ordering of the messages inside the WSDL operations is interpreted differently, depending on the type of the activity:

- In a one-way activity, the WSDL operation contains exactly one message – the *input* message. In a send activity (used in client tracks), this message is sent by the test framework and received by the BPEL process. In a receive activity (used in partner tracks), this message is sent by the BPEL process and received by the test framework.
- In a synchronous two-way activity, the WSDL operation contains at least two messages – an *input*, and an *output* (and possibly *faults*, see below). In a send-receive activity (used in client tracks), the first message is sent by the test framework and received by the BPEL process; the second message is sent back from the BPEL process to the test framework. In a receive-send activity (used in partner tracks), the first message is sent by the BPEL process and received by the test framework, the second message is sent back from the test framework to the BPEL process.
- In an asynchronous two-way activity, two one-way operations are involved, each containing exactly one message – an *input* message. The first operation corresponds to the call, and the second to the callback. In a send-receive activity (used in client tracks), the message inside the call operation is sent by the test framework and received by the BPEL process; the message inside the callback operation is sent by the BPEL process and received by the test framework. In a receive-send activity (used in partner tracks), the message inside the call operation is sent by the BPEL process and received by the framework, and the message inside the callback operation is sent by the framework and received by the BPEL process.

Note that despite the use of two operations, the overall message flow of a synchronous and asynchronous two-way operation is exactly the same. The following table gives a short overview of the message flow.

	One-way	**Synchronous/asynchronous two-way**
Client	Message is sent by the test framework, received by the BPEL process.	First message is sent by the framework, received by the BPEL process. Second message is sent by the BPEL process, received by the framework.
Partner	Message is sent by the BPEL process, received by the test framework.	First message is sent by the BPEL process, received by the framework. Second message is sent by the framework, received by the BPEL process.

Table 1: Message Flow

The following figure shows an example of an interaction of BPELUnit with a BPEL process. The first and last calls are part of a send-receive activity (simulating a client) and use the WSDL operations of the BPEL process, whereas the second and third calls are part of a receive-send activity (simulating a partner) and use the WSDL operations of the partner.

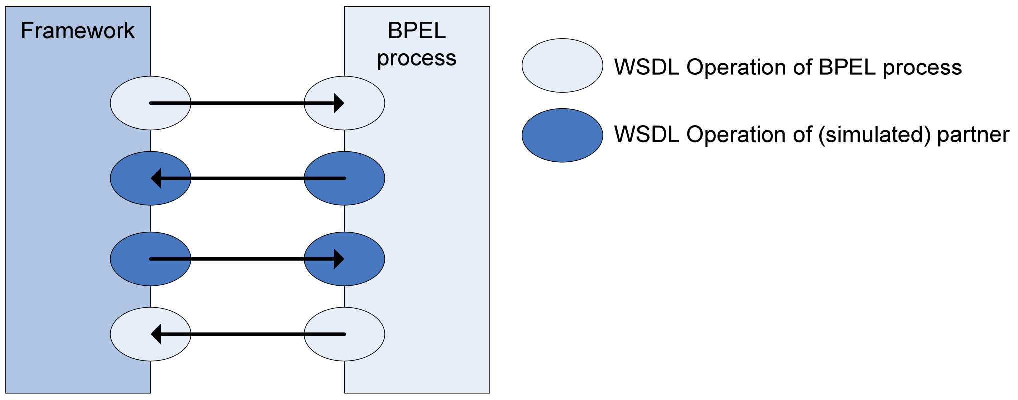

Framework
BPEL process
WSDL Operation of BPEL process
WSDL Operation of (simulated) partner

Figure 46: WSDL Operations in Use by the Framework

3.3.1.2 Fault handling

So far, only standard operations with input and output messages have been discussed. However, synchronous two-way operations also offer the ability to send faults. This ability must be propagated to the framework user. Therefore, the framework may additionally:

- **Receive faults**; i.e. instead of normal data, a SOAP fault is expected. This ability is normally used in a synchronous send-receive at the client side.
- **Send faults**, i.e. instead of normal data, a SOAP fault is sent. This ability is normally used in a synchronous receive-send at the partner side.

Note that one-way and asynchronous two-way operations do not offer the ability to send faults in BPEL, as WSDL does not allow the specification of faults in one-way operations. Special implementations, however, may differ here. The framework thus allows for such behavior, as it may come in useful when testing such implementations.

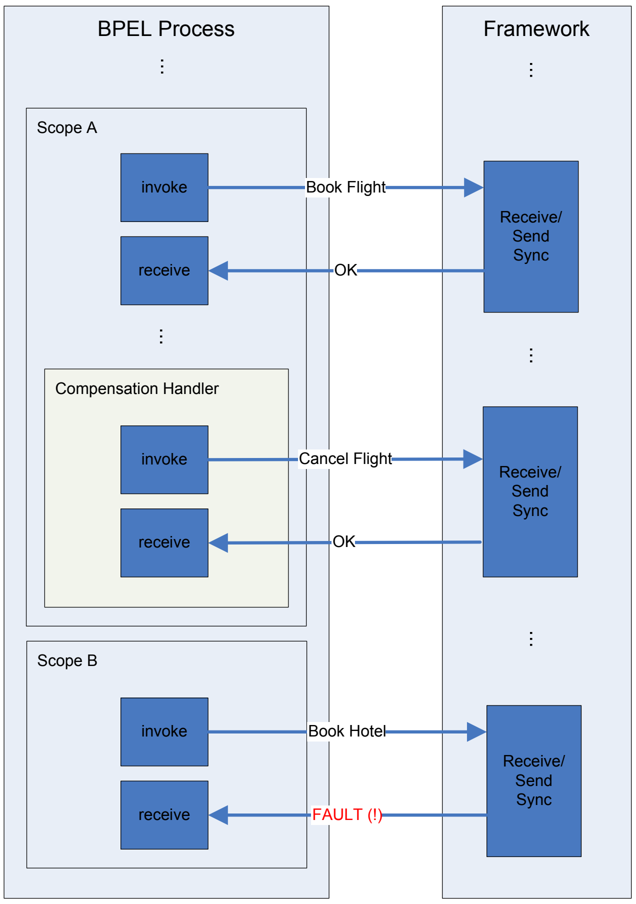


Figure 47: Causing Compensation

Fault handling is very important for the test framework, as it allows a tester to bring about compensation handling in a BPEL process. If such compensation handling is activated, the tester may verify the correctness of the compensation activities by means of the standard activities presented above, as BPEL compensation handlers also consist of a normal (BPEL) activity block.

Figure 47 shows an example of sending a fault from within the BPELUnit framework to activate a compensation handler in a previous scope. The BPEL process contains two scopes (A and B) included in a third (not shown). Scope A successfully books a flight; scope B tries to book a hotel. When scope B fails (a fault is specified to be sent by the tester), the enclosing scope invokes the compensation handler of scope A, thereby canceling the flight, which is again tested by the framework.

3.3.1.3 Testing flows

As pointed out at the beginning of this section, parallel behavior is built-in into BPEL. The test specification should offer some mechanisms for testing such behavior. In particular, the `flow` activity inside the BPEL language may leverage several receive activities, waiting for incoming messages of several partners at once. In this case, the framework should offer some means of specifying defined answer times for each partner in order to test all possible message arrival sequences.

BPELUnit offers such a possibility by introducing *delay sequences*. This feature allows the specification of an ordered list of *delay times* for each send operation. When a test case is executed, the framework creates *n* runtime test cases, n being the number of delay times in the sequences. In each run, the corresponding delay is introduced at each send operation, thus allowing very flexible configuration of message arrival times.

3.3.1.4 Correlation

Another important principle to consider is *correlation*. BPEL uses application-data for correlation, for example, a flight number included in a message to a partner. This only works if the returning message from the partner carries the same number as the outgoing message.

Values used for correlation are arbitrary, ranging from predefined values to values from a database to random numbers. Receive-reply activities such as the synchronous and asynchronous two-way operations supported in BPELUnit must have the ability to copy such values from the incoming message to the outgoing message in order to allow for correlation.

BPELUnit allows adding *data copy operations* to such activities. A data copy operation consists of two XPath expressions:

- **Source**. The source XPath expression identifies a node inside the incoming message which contains a correlation value to be copied.
- **Target**. The target XPath expression identifies a place inside the outgoing message which should carry the correlation value.

Copying values from one XML message to another has the same problems as copying values into XML variables in BPEL – the target may not yet exist. As in concrete BPEL implementations, BPELUnit creates the target element in such cases.

3.3.1.5 Specification summary

A PUT has one client and an arbitrary number (including zero) of partners. These Web services all run in parallel, interacting with the PUT. The interaction details specified as part of the test must thus also cover a number of parallel sequences of defined interactions with the PUT, all of which must be completed successfully for a test case to pass. If one of the interactions fails, for example if a condition does not hold or no call is received at all, the test fails and is aborted.

A BPELUnit *test case* thus consists of one client track and a number of partner tracks, which each specify a sequence of activities. While the client and partner tracks are all executed simultaneously, the activities in each track are executed sequentially. This has a number of implications.

- Specification of a test case is very straightforward. As the partner tracks all run in parallel, the developer can focus on the actual business protocol of each partner in isolation. Also, the sequence of activities is rather easy to follow.

- The specification of asynchronous calls is not very different from synchronous calls and thus, asynchronous calls can be tested in a very high-level and easy fashion.
- The test framework does not allow *nested* calls. Nested calls might be implemented on behalf of the partners or the BPEL service itself. It is still unclear whether nested calls will be a main pattern for the implementation of Web services.

The framework assists the tester in detecting the three kinds of incoming data errors listed in section 3.2.1. Incorrect message data is easy to detect: the XPath conditions in the receive activities take care of this. A missing message is detected through timers in the framework: If an expected message is not received within a certain time, a fault is generated. The case of too many messages is also handled by the framework: If it does not find a waiting receiving activity for a message, a fault is generated.

By using literal XML data and sequences of atomic interactions with the PUT, the BPELUnit test specification approach lies in-between the data-driven and logic-driven ends of the scale. As the test data corresponds directly to the XML Schema type definitions of WSDL message parts, the tester is able to operate *on the same data level* as if he were programming in BPEL. The test specification is thus a highly specialized mini-language directly aimed at rapid BPEL composition testing.

Due to the simplicity of the language, the actual test execution is not very difficult for the framework; automation is perfectly supported. Tool support for creating the test specification is also possible.

3.3.2 Test organization

As defined in the test specification, a test case consists of a number of parallel interaction threads describing the simulated business protocols of client and partners of the PUT. The test organization must be able to group these test cases into suites. Additionally, there are still some parts which cannot be integrated: The PUT itself, referenced WSDL files, and possibly other files (for example, XSD files with the data types). These artifacts are required by the test and must be linked from within the test organization.

The BPELUnit test organization combines multiple test cases with a setup/shutdown part containing links to the external artifacts, thereby creating a test suite. The test organization is thus logically separated from the test specification, but may still be combined in a common file. The resulting integrated test suite specification and organization document is called the *test suite specification*.

In BPEL composition testing, a setup and shutdown phase before and after the test case execution is mandatory, as partner Web services and the PUT itself need to be set up before the test, and later shut down again. The combined setup/shutdown part of the test suite contains all necessary information to execute these two phases.

A BPEL process normally has several partners; in most cases, a test case will therefore also contain partner tracks for these partners (note that in some test cases, the business protocol executed will skip some partners, therefore, the corresponding partner tracks are not needed in these tests). Several test cases may need to receive and send the same test data from and to some partners, only differing in other partners. For this reason, BPELUnit includes the concept of *test case inheritance*.

To inherit partner tracks from another test case, a test case may be declared as being *based on* another test case, thus creating a parent-child relationship. Each test case may only be based on one other test case (single inheritance). The child test case inherits all partner tracks from its parent which are *empty* in the child test case. Note that this also works over several inheritance levels.

Additionally, a BPEL test case may be declared *abstract*; this concept relates to abstract classes in conventional programming languages. An abstract test case is not executed, but may serve as a parent test case for other abstract or concrete test cases.

BPELUnit also supports augmentation of test cases with additional metadata, like for example the requirement tracing data mentioned in section 3.2.2.

Figure 48 gives an overview of the various parts of the test suite specification.

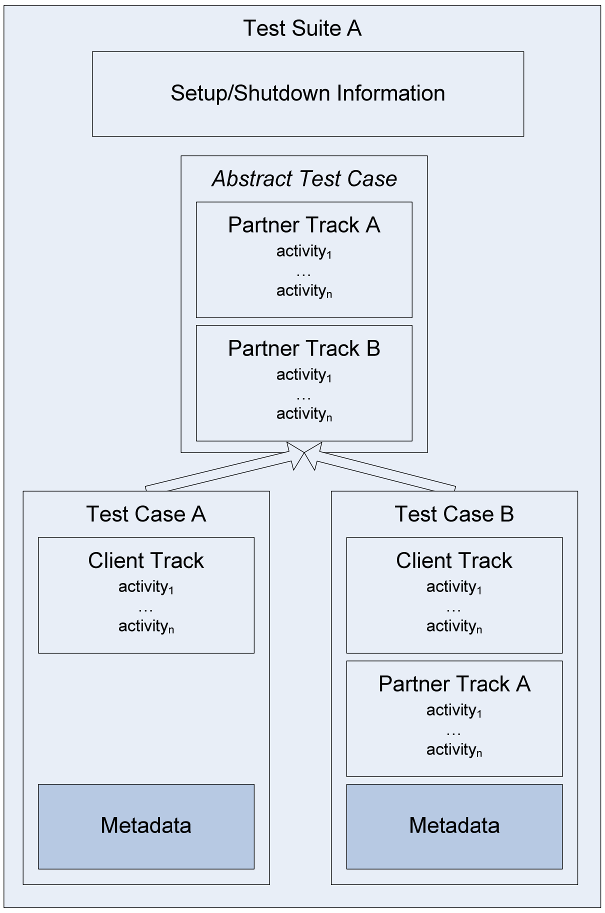


Figure 48: BPELUnit Test Suite Specification

The enclosing element corresponds to the complete test suite, which carries a setup/shutdown part, and a number of test cases. One test case is declared abstract, and two test cases (A and B) inherit from that test case. As the abstract test case did not declare a client track, this track is defined by both test cases. Test case A uses the partner tracks A and B from the abstract test case as-is, while test case B overrides the partner track A, specifying its own activities.

Both concrete test cases contain an additional metadata block which carries more information about the test case.

The test suite specification creates a central access point to the test, which is used as a starting point for test execution.

3.3.3 Test execution

Section 3.2.3 discussed two different options for executing the PUT to run a test – simulation and real-life testing. Both available choices are heavily dependent on some sort of API or deployment descriptor of the engine, which means vendor lock-in, as there are no standards in this area.

While simulated testing allows a more fine-grained control over the BPEL process instance, it binds a testing framework much closer to the actual engine in use, and also offers a much less realistic view on the BPEL runtime behavior, as the Web service stack usually lying beneath the BPEL implementation is completely bypassed. However, this stack may lead to subtle problems in BPEL compositions and should thus not be ignored.

Therefore, BPELUnit uses a real-life deployment approach: The process under test is deployed into an engine before the test and undeployed after the test. To enable such behavior, the framework must contain *adapters* for each engine, which talk to the engine and handle deployment.

Using real-life testing has a number of implications for the testing framework. First and foremost, BPELUnit needs to reach the BPEL process inside the BPEL engine on behalf of the client, and the BPEL process needs to reach the framework, as it also simulates the process partners. BPELUnit must thus *implement many parts of a Web service middleware* – in particular, calls made to the process and the calls received are encoded in SOAP; partners must be simulated by respecting their WSDL descriptions; and callback addressing must be employed to enable asynchronous calls.

As already mentioned, the Web service stack consists of various different protocols and combinations. Luckily, the BPEL specification already cuts down on some, for example, it is based on WSDL and XML Schema. Additionally, the WS-I Basic Profile makes some clear recommendations, which are followed by BPELUnit. Still, some protocols must be implemented to be able to talk to the BPEL process. BPELUnit implements the following services:

- SOAP handling for both rpc/literal and document/literal styles
- HTTP connectivity
- WS-Addressing callback addressing

Figure 49 gives an overview of the Web service stack needed by both the BPEL engine and BPELUnit.

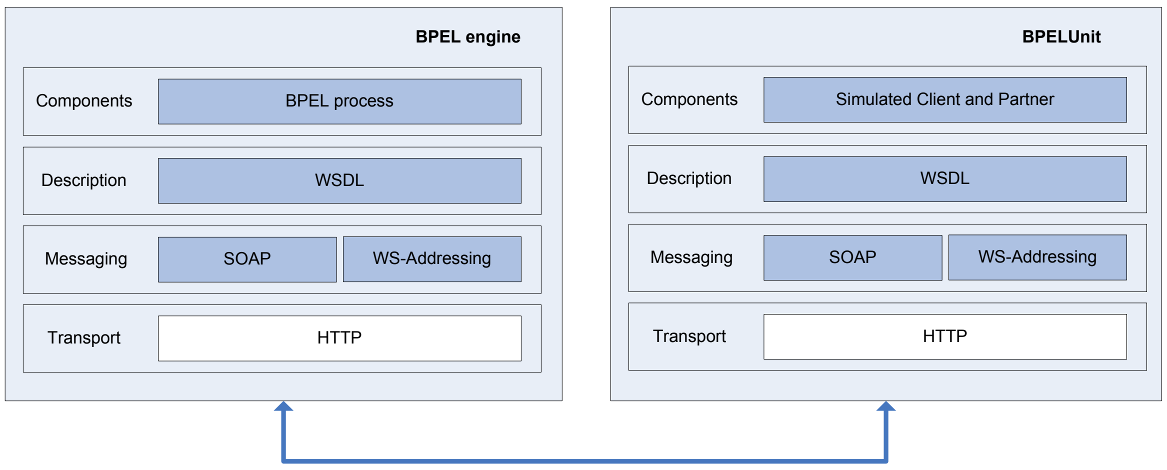


Figure 49: BPELUnit Middleware Implementation

On the left-hand side, the implementation of the BPEL engine is displayed, running the BPEL process as the top-level component. The BPEL engine implements at least WSDL, SOAP, WS-Addressing, and HTTP for external communication.

On the right-hand side, BPELUnit is displayed, which uses simulated clients and partners as components on top of WSDL, SOAP, WS-Addressing, and HTTP to be able talk to the BPEL process.

The next three subsections discuss three important points to consider in this architecture.

3.3.3.1 SOAP calls and encoding

As pointed out in section 3.3.1, BPELUnit uses BPEL-level data (i.e. plain XML) for specifying data to be sent to the process under test, and as a basis for the XPath receive conditions. However, when using real-life testing, the framework must follow the usual standards when talking to the BPEL process, i.e. respect the WSDL descriptions and send and receive SOAP messages.

To achieve this, BPELUnit reads the WSDL concrete part of both the BPEL process itself and of partners to extract the SOAP style and encoding required by a particular call. Each BPELUnit activity (for example, a synchronous receive-send) must therefore be *bound* to a certain WSDL operation. BPELUnit uses this information to correctly encode and decode the SOAP data sent to and received from the BPEL process.

This allows the developer to remain blissfully ignorant of the SOAP encoding problems lying beneath the testing level – until something goes wrong. In this case, the fact that the framework has complete control over both partners and clients is of great advantage, as every communication artifact can be recorded by BPELUnit and presented to the user, who can then debug the implementation to determine the reason and origin of the faulty behavior.

3.3.3.2 Partner simulation

As the BPEL process is executed normally, it expects its partners to be full-blown Web services themselves. The framework must thus simulate the partners as Web services.

To achieve this, BPELUnit contains a HTTP server through which it is able to bind incoming calls to a concrete simulated partner, and in particular, an activity specified inside the partner track.

3.3.3.3 Asynchronous callbacks

As the BPEL process may offer asynchronous operations to clients, or invoke them on partners, the framework must be able to handle callback addressing information in order to enable callbacks from the BPEL process, and create callbacks itself.

As pointed out in chapter 2, callback addressing is decoupled from SOAP. The WS-Addressing standard has been presented as a remedy, and is used by some BPEL vendors, for example, Oracle and ActiveBPEL. Callback addressing has different requirements on the client and partner side:

- On the client side, the framework must augment SOAP calls with addressing information to allow callbacks from the BPEL process itself, i.e., the callback information is created by the framework and read by the BPEL engine.
- On the partner side, the BPEL engine creates the addressing information and augments SOAP calls to the partners with that information, i.e., the callback information must be read by the framework and used to create a callback on behalf of the simulated partner.

BPELUnit transparently attaches and extracts addressing information to and from the messages sent and received where required.

3.3.4 Test results

Using real-life deployment within the framework makes it difficult to gather any information on what is going on inside the tested PUT instance. As pointed out before, one way of gathering such information would be to include specific logic for this in the test processes. However, such an approach would never yield complete results.

Another way would be to use available APIs to query the engine about the state of a process instance after its completion. However, such API is highly vendor-dependent.

Therefore, BPELUnit only follows the basic xUnit approach, reporting on successes, failures, and errors. BPELUnit categorizes failures and errors as follows:

- A **failure** is an application-level error: For example, an expected message from the BPEL process is missing; contains the wrong information; or too many messages are received.
- An **error** is a problem with the Web service stack: The server does not respond or replies with the wrong error codes; the messages do not contain XML; or other communication level problems.

Adding metrics support, however, still remains an interesting problem. An outlook on possible further research is given in chapter 6.

3.3.5 Conclusion

BPELUnit is a unit testing framework for repeatable, white-box, automated BPEL composition testing, which lays particular emphasis on the following points:

- Support for BPEL-specific items like parallel activity handling, compensation, and asynchronous messaging.
- Ease of use for the developer.
- A realistic view on the execution of a BPEL process.

These points are realized by employing a BPEL-level data specification format, a domain-specific language for writing the test logic, a test suite format which allows re-use of test cases, and real-life BPEL deployment and execution.

3.4 Existing approaches

As the BPEL language is relatively new, there are – so far – not many efforts for creating unit testing frameworks specifically targeted at BPEL. The area of BPEL composition testing is currently restricted to one theoretical approach (Li et al. 2005) and several practical approaches embedded into BPEL IDEs (for example in the ActiveBPEL designer, or Sun's NetBeans).

The theoretical approach in Li et al. 2005 contains some initial ideas on BPEL unit testing. It uses BPEL as the test specification language, requiring testers to create a BPEL test process for each partner of the PUT as well as a central, coordinating process. Testing BPEL with BPEL is an interesting approach, especially in lights of the xUnit family.

However, the paper does not contain information about how to actually run the tests (as BPEL itself does not allow user interactions), how to deploy the processes, and on the particular problem of parameterizing the BPEL test mocks, i.e. instructing the mocks what data to expect and send in a particular test case. The BPELUnit approach presented in this thesis differs from Li et al. 2005 in that it goes a step further by clearly addressing test parameterization, organization, and execution.

Existing practical approaches built into IDEs fall into two categories. In the first category, a simple black-box approach is used, i.e. data is sent into a BPEL process and an answer is expected (for example, in the Oracle BPEL Console). This means that the BPEL process is not tested as a composition, but as a simple Web service, which is a different setup. In the second category, the BPEL engine runs in a “simulation mode”, which allows the testing framework to directly inject or extract data instead of making actual SOAP calls (for example, in the ActiveBPEL Designer). However, these approaches offer only manual testing, are limited to a particular engine, and bypass the Web service stack.

As BPEL processes are Web services, existing Web service testing tools can also be used for BPEL composition testing. However, these tools either regard Web services as black boxes (only simulating a client), or are intended to test Web service clients (i.e., simulate a Web service). Examples of tools from these two classes are the two open-source implementations ANTEater (Predescu et al. 2006) and WS-Unit (Bradby 2006). ANTEater is a functional testing tool intended for simulating a client. WS-Unit, on the other hand, is a “Web service consumer tester”, i.e. it can be used to simulate a BPEL partner.

However, a BPEL unit testing framework must include functionality for simulating both partners and clients of a BPEL process at the same time. BPELUnit provides such an integrated solution, enabling the simultaneous tracking of the progress of the client and all partners in one homogenous environment. Additionally, BPELUnit differs in three important points from the tools mentioned above:

- BPELUnit uses literal XML data as the data specification format instead of complete SOAP envelopes, which puts the tester on the same level with the BPEL compositions.
- BPELUnit allows the specification of parallel threads of sequential activities required for simulating several partners of a PUT at once, each fulfilling a certain business protocol.
- BPELUnit provides the ability to extract callback information to create server-side callbacks.

All in all, it is believed that the BPELUnit framework presented in this thesis is currently unrivaled in its abilities for testing BPEL processes.

4 Design and implementation

This chapter discusses the (technical) design and implementation of BPELUnit. BPELUnit consists of two major components: the *framework*, and the *tool support*.

- **Framework**. The BPELUnit framework implements the actual testing framework, i.e., partner simulation, invocations of the process under test (PUT), and result gathering.
- **Tool Support**. The BPELUnit tool support consists of utilities which ease the development of unit tests by providing an UI for writing the test suite specification document.

In section 4.1, the basic design considerations for the framework and tool support are presented on a high level, which are used for the concrete software design in later sections.

Section 4.2 then discusses the design and implementation of the BPELUnit framework – the test specification language, the framework core, extension points, clients, and the integration of clients into other frameworks.

In section 4.3, the BPELUnit tool support is presented, which provides a UI for creating test suite specification documents.

4.1 Design considerations

This section introduces the basic requirements for the BPELUnit components. First and foremost, the framework should be independent of any concrete runtime environment; i.e., it should be possible to run BPELUnit tests on a developer machine (for example, as part of an IDE, with a graphical user interface), and on a server (for example, a headless development machine which creates nightly builds and runs all unit tests as part of the build).

This requirement basically necessitates a separation of a framework *core*, which handles the actual test runs, and *clients*, which offer the functionality to the user. The core must thus provide an API for the implementation of arbitrary clients.

Secondly, BPELUnit uses a real-life testing approach, which means the PUT must be deployed into an engine before the test, and undeployed afterwards. BPELUnit should not limit itself to particular engines and must therefore be extensible to allow arbitrary BPEL engine deployers.

Thirdly, BPELUnit has to deal with a complicated and extensible Web service stack. The BPEL process and partners may be implemented using arbitrary styles and encodings. Being based on the WS-I Basic Profile, BPELUnit requires WSDL descriptions, and SOAP transfer via HTTP. However, with regard to the concrete style and encoding of the SOAP binding as well as the mechanism used for callback addressing, the framework should allow extensions to plug in additional mechanisms.

The BPELUnit core thus needs to be designed to be easily extensible for:

- New BPEL deployers, which deploy and undeploy a process to and from an engine.
- New SOAP encoders/decoders, which handle a particular style/encoding combination.
- New callback handlers, or more generally, SOAP header processors.

Finally, BPELUnit needs to specify a concrete language, which the developer can use to create the test suite specification documents. For writing these documents, a graphical editor should be available to facilitate testing; this editor (the *tool support*) should be a full replacement for a source editor.

4.2 BPELUnit framework

The *framework* implements all aspects of BPELUnit which were discussed in chapter 3.3. The following sections will present both design and implementation of these aspects.

Consistent with many BPEL engines and the goal of platform-independence, the BPELUnit framework is written in Java. The tasks of the framework include reading the test suite specification, deploying the linked BPEL process, starting and managing the partner tracks, and gathering results from the tracks.

Figure 50 shows the design of the framework. The core provides an API and three extension points to extenders.

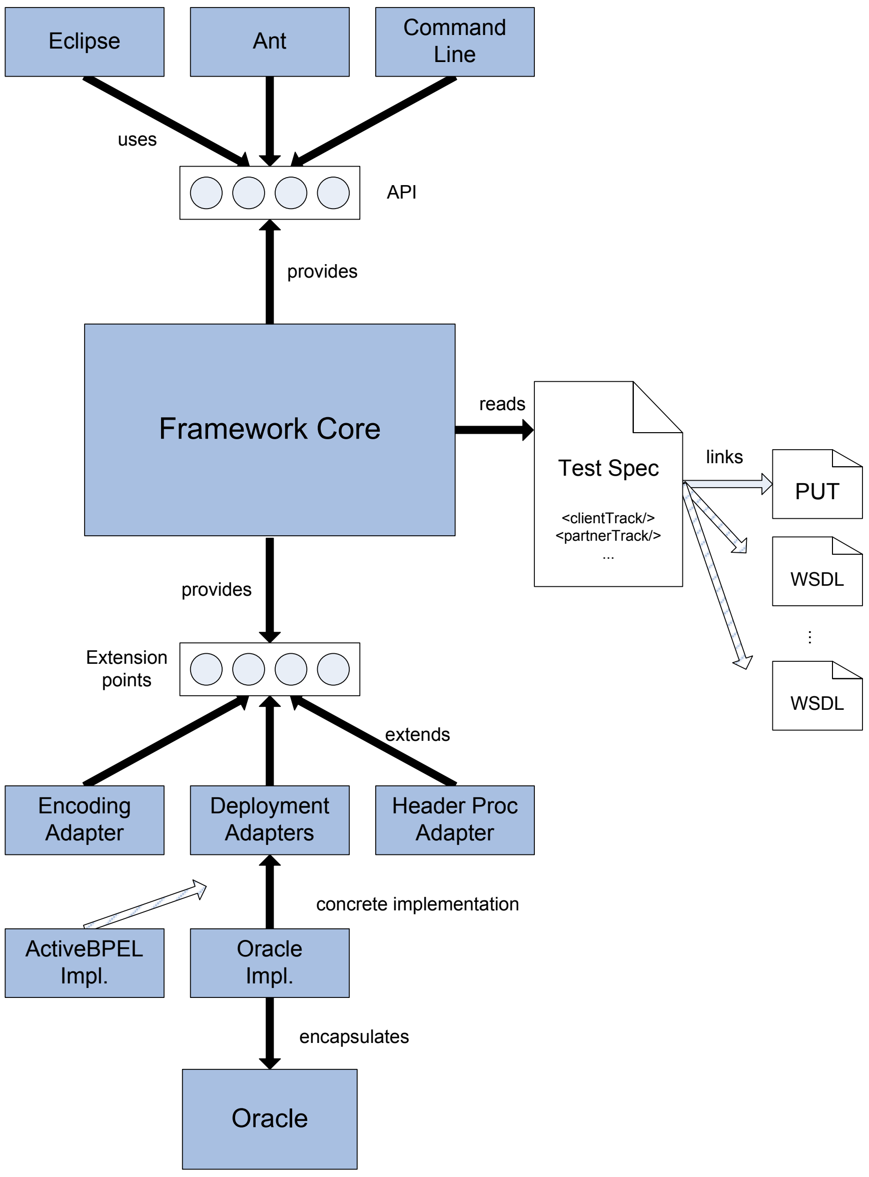


Figure 50: The BPELUnit Framework

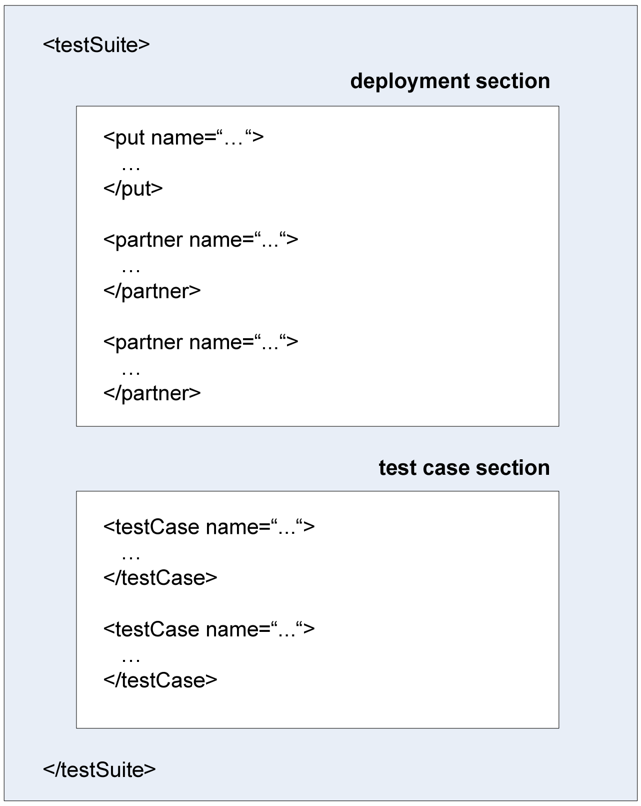
The following four major components of the BPELUnit framework will be discussed in the subsequent sections:

- On the right hand side of the core, the *test (suite) specification* is displayed. This document is written in XML; the concrete format is discussed in the next section.
- In the middle of the diagram, the *framework core* is displayed. The implementation of the core itself will be detailed in section 4.2.2.
- At the bottom of the diagram, the three *extension points* of BPELUnit are displayed: SOAP Encoding, BPEL Deployment, and Header Processors. An example is given of two deployment adapters, one for ActiveBPEL, and one for Oracle. These extension points and the two standard implementations of BPELUnit (deployers for ActiveBPEL and Oracle) will be discussed in section 4.2.3.
- Finally, the top of the diagram shows an *API* available to clients. As an example, clients for the Eclipse IDE, the Ant build tool, and the command line are displayed. Section 4.2.4 will give an overview about how to use this API, and the clients which are already included in BPELUnit.

4.2.1 Test specification

The easiest way for integrating test data, interactions, and deployment information of a test suite is to use an XML-based format. BPELUnit therefore employs an XML-based test suite document, which includes a setup- and shutdown part and the test cases. The default file ending for the test suite documents is `.bpts` for *BPEL Test Suite*. An XML Schema is provided to validate these documents.

A test suite document consists of two parts, as shown in Figure 51.



```
<testSuite>
                                   deployment section
   <put name="…">
     …
   </put>

   <partner name="...">
     …
   </partner>

   <partner name="...">
     …
   </partner>

                                   test case section
   <testCase name="...">
     …
   </testCase>

   <testCase name="...">
     …
   </testCase>

</testSuite>
```

Figure 51: The Test Suite Document

The first part corresponds to the setup/shutdown section discussed in the previous chapter. This part is called the *deployment section*, as it contains the information for deployment of the PUT, and specification of all the partners to be simulated.

The second part is the *test case section*, which contains an arbitrary number of test cases, each in turn containing the client and partner tracks with activities.

The `testSuite` root XML element itself contains two additional elements (Figure 52):

- **Test Suite Name**. This is a human-readable name of the test suite and will be presented as part of all test result outputs.
- **Base URL**. BPELUnit simulates partners at a local URL. The URL of each partner is a concatenation of the base URL given as part of the test suite, and the partner name. The default base URL is `http://localhost:7777/ws/`, which may be changed to any (local) address.

```
<testSuite>
	<name>MetaSearchTest</name>
	<baseURL>http://localhost:7777/ws</baseURL>
	...
</testSuite>
```

Figure 52: Test Suite Specification

4.2.1.1 The deployment section

In the deployment section, developers specify the PUT data and all partners of the PUT to be simulated by the framework. The deployment section is enclosed in the `deployment` XML element.

The specification of the PUT basically looks like Figure 53; it contains at least:

- A **name**. The name is used by the deployer of the PUT, and for later reference in the test case section if needed.
- A **type**. The type of the PUT is the deployment type, i.e. identifies the deployer which deploys the PUT. As noted before, BPELUnit allows the introduction of arbitrary deployers by means of an extension point, which will be discussed in later sections.
- A **WSDL file**. As the framework needs to create calls to the PUT targeted at the correct address and with correct SOAP encoding, a link to a complete WSDL specification of the PUT is required.

```
<put
     name="MetaSearch"
     type="oracle">

     <wsdl>MetaSearch.wsdl</wsdl>

     <property name="..."> ... </property>
</put>
```

Figure 53: PUT Specification

Besides the three items mentioned above, the PUT specification may also contain an arbitrary number of property-value pairs. Each property name is a configuration option for the concrete PUT deployer, allowing flexible configuration of each deployer according to its own requirements. PUT deployers will be discussed in section 4.2.3.1.

In addition to the PUT itself, the deployment section also contains the specification of partners of the PUT. Note that a PUT may not have any partners, the partners may not be required for the test, or need not be simulated; therefore, specification of partners is optional. Without any partners specified, BPELUnit acts as a black-box testing tool.

Each partner is specified with:

- A **name**. The name identifies the partner for later use in the test case section, and also forms the second part of the simulated URL of this partner.
- A **WSDL file**. The WSDL file will be read from a local file system path and must contain the complete Web service specification of the partner.

The URL of the simulated partner is very important and must be included by the developer himself in the WSDL file of the partner, such that the BPEL engine running the PUT can find the simulated partner at the designated location. For example:

- **Base URL**: `http://localhost:7777/ws/`
- **Partner name**: `airline`
- **Simulated partner URL**: `http://localhost:7777/ws/airline`

Figure 54 gives an example of the specification of a partner inside the `deployment` element. This concludes the description of the deployment section.

```
<partner
     name="Airline"
     wsdl="TravelAirlineReservation.wsdl"
     />
```

Figure 54: Partner Specification

4.2.1.2 The test case section

The test case section contains the actual test cases of a test suite. Each test case is enclosed in a `testCase` XML element and contains one client track (which is required), and a number (including zero) of partner tracks.

Figure 55 contains an example of a `testCases` section with one test case. The test case contains one client track and one partner track for the partner `Airline` from Figure 54.

```
<testCases>
     <testCase name="Travel Test">
          <property name="useCase">245</property>

          <clientTrack>
               ...
          </clientTrack>

          <partnerTrack name="Airline">
               ...
          </partnerTrack>
     </testCase>
</testCases
```

Figure 55: Test Cases Section

As can be seen in the figure, BPELUnit also defines a property/value-based extension mechanism for each test case, which can be used to define arbitrary properties and their values (like the use case specification in the example) to add test management capabilities. These remain untouched by the framework, but are provided via API to clients.

Each test case has a name, which is specified as an attribute on the `testCase` element. The name is used for two purposes:

- It is used by the framework to identify the test case, presenting the results to the user.
- It may be used in other test cases to specify a parent test case.

Besides the name, the test case may also specify two attributes which are relevant for test case inheritance:

- `basedOn` attribute. The `basedOn` attribute is used to specify another test case on which the current test case is based as discussed in chapter 3.
- `abstract` attribute. If true, this test case is regarded as being abstract and will not be executed.

Figure 56 gives an example of a non-abstract test case, which inherits from another test case.

```
<testCase
     name="ExpireOnceTest"
     basedOn="NonEscalationTest"
     abstract="false"
     />
```

Figure 56: An Inheriting Test Case Specification

As pointed out above, each test case must contain one client track and a number of partner tracks. Each `partnerTrack` element must contain a link to one of the partners specified in the deployment section, and each test case may only contain one partner track for each partner.

Besides these differences, the contents of the tracks are exactly the same; each consisting of a sequence of activities. As discussed in the previous chapter, BPELUnit supports six activities:

- `sendOnly`. A send-only activity sends a single message.
- `receiveOnly`. A receive-only activity waits for a single message and verifies it.

- `sendReceive`. A send-receive synchronous activity sends a single message, waits for a synchronous response, and verifies the response.
- `receiveSend`. A receive-send synchronous activity waits for a single message, verifies it, and sends back a synchronous response.
- `sendReceiveAsync`: A send-receive asynchronous activity sends a single message, waits for an asynchronous response, and verifies the response.
- `receiveSendSync`. A receive-send asynchronous activity waits for a single message, verifies it, and sends back an asynchronous response.

An activity may contain various parts, depending on its type and use. Some of the parts only make sense in special types of activities; some are required in all activities:

- **WSDL operation**. Each activity needs at least one link to a WSDL operation, which is specified by the qualified name of the WSDL service, the WSDL port name, and the WSDL operation itself. The two asynchronous operations need two operation specifications (different ones for send and receive).
- **Send block**. Each activity which sends something (i.e., every activity except `receiveOnly`) needs a send block which contains the literal XML data to be sent.
- **Receive block**. Each activity which receives something (i.e., every activity except `sendOnly`) needs a receive block, which contains the conditions to be checked on the incoming message.
- **Data copy block**. Each activity may have a data copy block, which contains copy operations for copying data from input- to output message. This block only makes sense in the two receive-send activities.
- **Header processor block**. Each activity may declare a header processor, which processes the SOAP header immediately before the message is sent, and immediately after it is received, respectively. A header processor is required for asynchronous operations, as these require special callback addressing, but may also be used for other activities if required by the BPEL engine.

Three examples are given, each for one of the activity types (one-way, synchronous two-way, and asynchronous two-way).

Figure 57 gives an example of a send-only activity. Note that the WSDL operation is specified directly at the root node of the `sendOnly` element. The `data` element inside the `sendOnly` element then carries the literal XML data to be sent within its own namespace (which may be declared on the element itself, or somewhere else).

```
<sendOnly
     service="wpb:WastePaperBasketWebService"
     port="WastePaperBasketWebServicePort"
     operation="initiate">
     <data>
          <wpb:WastePaperBasketUsage>
               <wpb:WasteThrower>Philip</wpb:WasteThrower>
               <wpb:WasteContent>Good Intentions</wpb:WasteContent>
          </wpb:WastePaperBasketUsage>
     </data>
</sendOnly>
```

Figure 57: A Send-Only Activity

Figure 58 contains a send-receive synchronous activity. Note that again, the WSDL operation is specified at the root element, as a synchronous two-way activity corresponds to one WSDL operation.

However, this activity does not only contain a `send` block, but also a `receive` block. The receive block specifies one `condition` which must hold true for this activity to pass the test. The condition consists of an XPath `expression` to be executed inside the received literal XML data, and a `value` which must correspond to the result of the XPath expression.

The *expression* must be addressed relatively to the root of the literal XML data of the incoming message. The *value* may also be any valid XPath statement.

```
<sendReceive
     port="BookingProcessPort"
     operation="process"
     service="client:BookingProcess">

     <send>
          <data>
               <client:bookme>
                   <client:employeeID>848</client:employeeID>
               </client:bookme>
          </data>
     </send>

     <receive>
          <condition>
               <expression>
                   client:bookinganswer/client:booked/text()
               </expression>
               <value>'true'</value>
          </condition>
     </receive>

</sendReceive>
```

Figure 58: A Send-Receive Synchronous Activity

Finally, the next figure gives an example of an asynchronous receive-send activity.

```
<receiveSendAsynchronous>

     <headerProcessor name="wsa"/>

     <receive
          service="aln:TravelAirlineReservation"
          port="TravelAirlineReservationPort"
          operation="initiate">

          <condition>
               ...
          </condition>
     </receive>

     <send
          service="aln:TravelAirlineReservationCallbackService"
          port="TravelAirlineReservationCallbackPort"
          operation="onResult">

          <data>
               ...
          </data>
     </send>

</receiveSendAsynchronous>
```

Figure 59: A Receive-Send Asynchronous Activity

A few things are to be noted in comparison to the synchronous send-receive:

- There are two WSDL operations: The first is attached to the `send` sub element, the second to the `receive` sub element.
- The activity contains a header processor specification.

A header processor is specified with a name, and (optionally) a number of property-value pairs. The name is a link to a registered header processor, of which there might be many (see section 4.2.3.3); the property-value pairs may be used to instruct the header processor if necessary.

The selected processor (`wsa`) does not require special instructions; therefore, the example does not contain any property-value pairs.

As noted above, an activity may also contain an optional data copy block. The following figure shows such a data copy block, which has a `mapping` root element (as data copying is intended for mapping correlation data) and an arbitrary number of copy operations inside, copying data from an XPath-addressed point in the incoming message to an XPath-addressed point in the outgoing message.

Both XPath expressions are relative to the root of the literal XML data. A data copy specification only makes sense for receive-send operations (synchronous or asynchronous).

```
<mapping>
   <copy
     from="aln:TravelAirli...vationProcessRequest/aln:class[1]"
     to="aln:TravelAirli...vationProcessResponse/aln:TravelClass[1]"
   />
</mapping>
```

Figure 60: A Data Copy Block

As discussed in the previous chapter, BPELUnit contains special support for testing BPEL `flow` activities by introducing delay sequences for send operations. To be able to use delays in a test case, the test case must be declared as being *varying*, like in the following figure:

```
<testCase
     name="ExpireOnceTest"
     vary="true"
     />
```

Figure 61: A Varying Test Case

Once a test case has been declared to be varying, BPELUnit creates several instances of this test case at runtime, in which each send activity is delayed differently, according to a sequence of delay times specified at each send activity. Not all send activities need to specify delays; however, those that do must specify exactly the same number of delays in a comma-separated list of delay times in seconds.

In the following send specification, two delays are introduced; the first one is zero (no delay), and the second one is 2 (two seconds delay). Another send in a different partner track would probably declare the inverse sequence.

```
<send
     delaySequence="0,2">

     <data>
          ...
     </data>
</send>
```

Figure 62: A Send Specification with a Delay Sequence

Finally, both receive and send specifications may declare awareness for *faults* by adding the attribute `fault`.

- In a send specification, the fault declaration indicates that the payload inside the data element is in fact a fault and should be sent as a SOAP fault (detail block).
- In a receive specification, the fault declaration indicates that a fault is to be expected (SOAP fault element, and in case of a synchronous receive, an HTTP error code of 500), and the conditions are to be tested against the fault detail block.

Figure 63 shows an example of a send element with `fault` set to `true`.

```
<send
     fault="true">

     <data>
          ...
     </data>
</send>
```

Figure 63: A Send with a Fault

With the introduction of fault awareness, the test specification has been completely described. The next section offers information about the implementation of the BPELUnit core.

4.2.2 Core implementation

The BPELUnit core is the software component responsible for reading the test specification, running a test suite, and gathering information from the run for later presentation.

In the following sections, three interesting aspects of the BPELUnit core are discussed:

- **Test artifact hierarchy**: Ranging from the test suite down to the concrete send and receive specifications.
- **Implementation of core classes**: Realization of the classes doing the actual work.
- **Result storage and error tracking**: Enabling users to view the full history of what happened in a test run.

Figure 64 gives an overview of the steps the core takes when invoked with a test suite document:

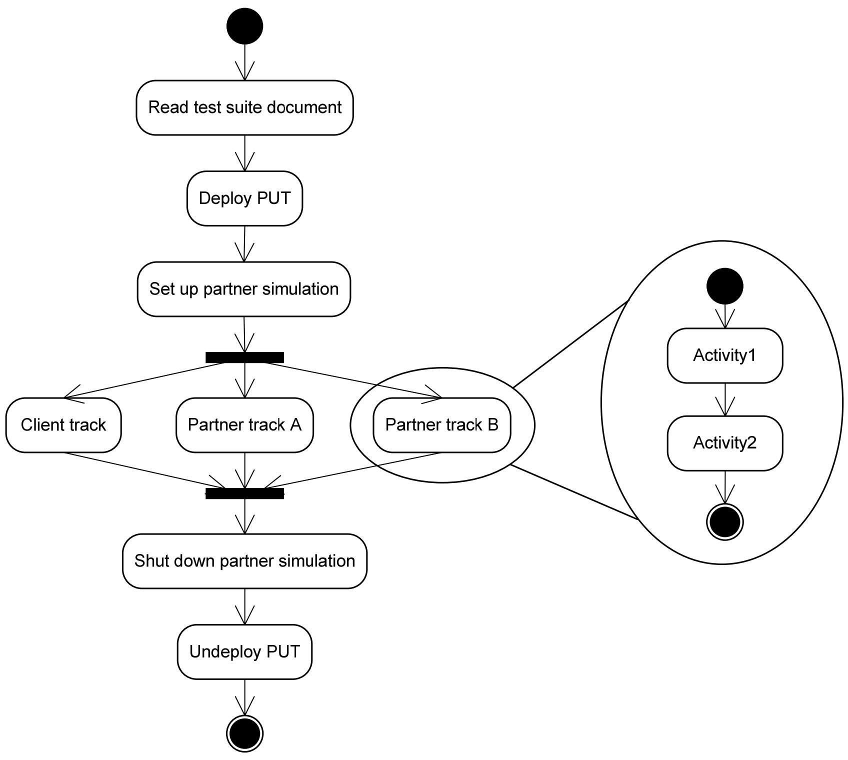


Figure 64: BPELUnit Core Implementation

The diagram shows three different phases:

- At the beginning, setup routines read the suite document, deploy the PUT, and initialize the partners.
- The main part of the action occurs within the parallel *tracks*, which run the activities.
- At the end, the PUT is undeployed and the partner simulation shut down.

4.2.2.1 Test artifact hierarchy

The model of BPELUnit consists of an internal representation of the test suite, test cases, tracks, and activities for a test run. The data contained within the model is initially read from the XML test suite specification file, and later augmented with run-time data. Each piece of the model, ranging from the suite to the activities and even lower, is called a *Test Artifact*.

The test artifact hierarchy is a composed tree of artifact classes, all implementing the common interface `ITestArtefact`.

The hierarchy below (Figure 65) shows the first part of the basic building blocks of the framework: `TestSuite`, `TestCase`, `PartnerTrack`, `Activity`, and all kinds of subclasses of `Activity`. The classes implement the respective functionality of their names.

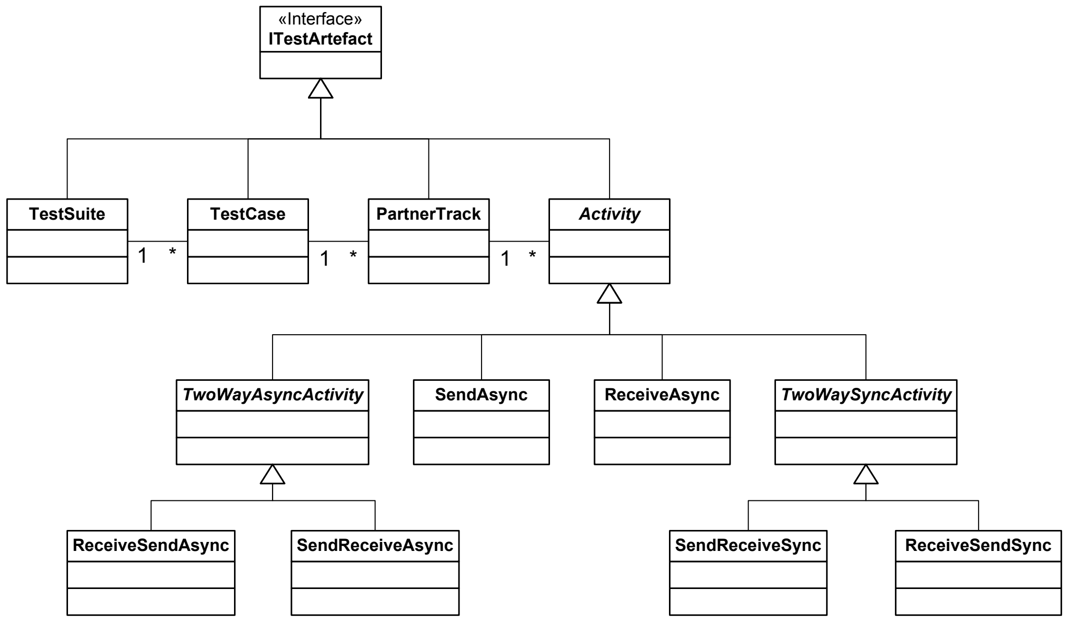


Figure 65: Framework Core Class Diagram (1/2)

The next diagram shows the more low-level elements (called *handlers*), which are contained within activities, and also implement `ITestArtefact`:

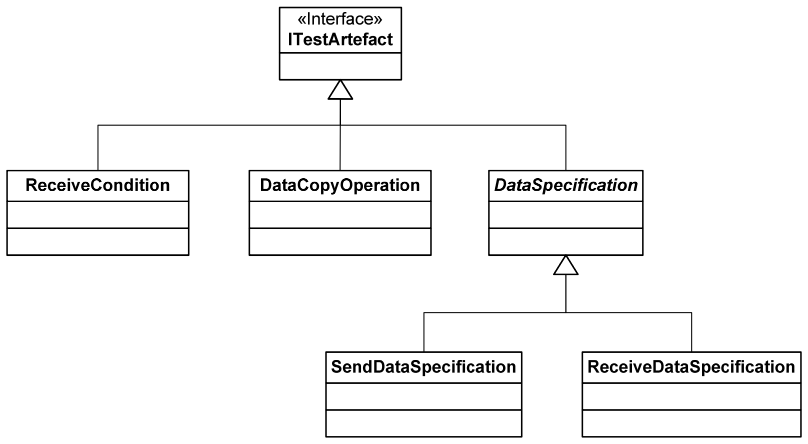


Figure 66: Framework Core Class Diagram (2/2)

The handler classes have the following functionality:

- `ReceiveCondition` implements a condition inside a receiving activity (XPath addressing, and comparison with a specified value).
- `DataCopyOperation` implements a copy operation inside of a two-way activity, copying data from input to output message.

- `DataSpecification` is the abstract base class for both `SendDataSpecification` and `ReceiveDataSpecification`. These two subclasses implement the send- and receive-blocks of all activities.

The handlers have been separated from the activities as all activities need them, but do so in very different ways. For example, a receive-only activity does not need a send specification or a data copy operation; a two-way receive-send operation needs everything. The following diagram clears up the relationships between the activity classes and the handlers:

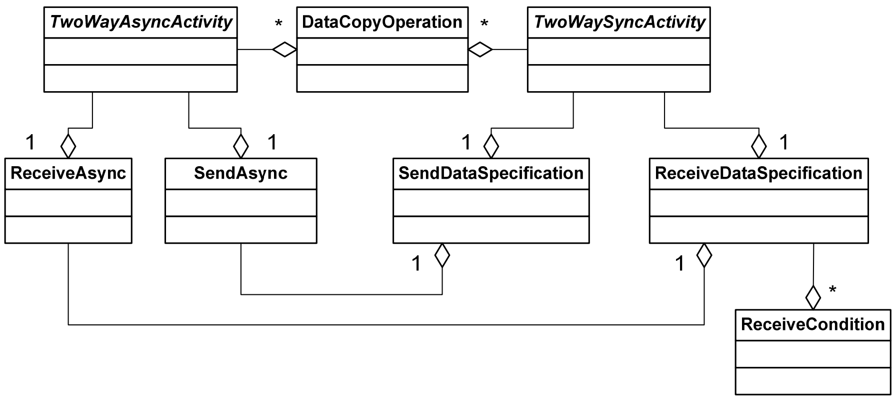


Figure 67: Relationship between Activities and Handlers

Several things are to be noted:

- First of all, a two-way asynchronous activity is internally implemented by employing the two one-way activities `ReceiveAsync` and `SendAsync`, as there are two transmissions.
- The `ReceiveAsync` and `SendAsync` activities use the `SendDataSpecification` and `ReceiveDataSpecification` handlers.
- The two-way synchronous activity, on the other hand, directly uses the specifications, as there is only one transmission.
- A `ReceiveDataSpecification` has one or more associated `ReceiveConditions`, which are used to check the user-defined receive XPath conditions.
- Both of the two-way operations have associated `DataCopyOperation` instances.

The next section discusses how these classes are executed in order to run a test.

4.2.2.2 Implementation of core classes

A test run is started by a client of the framework, which may be implemented in various ways (see section 4.2.4). As a result of the initialization by the test runner, a fully resolved `TestSuite` object is available, which already includes all composed classes for running the test (i.e., test cases, tracks, activities, and so on).

In its `setUp()` method, the `TestSuite` handles setting up the environment necessary for the test. This consists of the following two steps:

- The suite instantiates a deployer class for the PUT according to the type of the PUT given in the test specification, and instructs it to deploy the PUT.
- The suite starts the local HTTP server to be able to receive incoming calls for simulated partners (and/or the client).

Afterwards, a `run()` method sequentially runs all test cases in the suite. Among other things, test cases carry a list of client and partner tracks to be run, which are already resolved as per inheritance and/or test case variance.

Finally, a `shutDown()` method undeploys the PUT and shuts down the HTTP server.

When a test case is started, it creates a new instance of a utility class called the `TestCaseRunner`. This class is responsible for handling all thread-related tasks. In particular, the following three tasks lie within the domain of the runner:

- A new thread is created and started for each client and partner track (in the following, both are called partner tracks for simplicity).
- The runner waits for all partner tracks to finish their business and return.
- The runner aborts active partner tracks if something went wrong (failure/error) in another partner track.

All partner tracks report their progress to the runner (i.e., whether an activity has been completed, or the whole track completed with a success, an error, or a failure). The runner is also the *communication hub* for all messaging-related activities, connecting the HTTP server, the test cases, and the partners. The hub offers the following functionality:

- A partner track may ask the runner if an incoming message has been received for the partner.
- A partner track may tell the runner to send back a return message for a previous incoming request.
- A partner track may instruct the runner to synchronously send a message, and return the result to the partner track.

Each partner track consists of a series of *activities*, which are executed by the partner track in sequence. The activity classes are self-contained and carry out the logic in sequence, too. The `Activity` base class contains an abstract `run()` method, which is invoked to trigger execution of an activity.

This method is implemented in the various subclasses of `Activity`. Four of the classes implement the method themselves (`SendAsync`, `ReceiveAsync`, `SendReceiveSync`, and `ReceiveSendSync`), while two only delegate to other activities (`SendReceiveAsync` and `ReceiveSendAsync`).

The implementation of the first four activities is sketched out below. The activities basically handle sending and receiving of messages and checking for severe errors. The actual SOAP header handling, SOAP encoding and decoding, data copying, and receive condition checking is done by the two specification classes `SendDataSpecification` and `ReceiveDataSpecification`, which are described thereafter.

Figure 68 shows the activity diagrams for the send-only and receive-only activities. Both of these activities correspond to a one-message WSDL operation. A one-way WSDL message transported via SOAP over HTTP must be acknowledged by the server with a 2XX return code (mostly, 202 ACCEPTED).

Note that in the case of the receive-only activity, a return 2XX is generated by the framework regardless of whether a fault occurred. One-way WSDL operations are – per definition – not allowed to signal faults back to the caller.

For simplicity, this and the following diagrams do not differentiate between failures and errors. After completion, an activity is marked with a result code. The result is propagated to the outer partner track, test case, and finally, to the test suite, and is available via API to clients (see section 4.2.4).

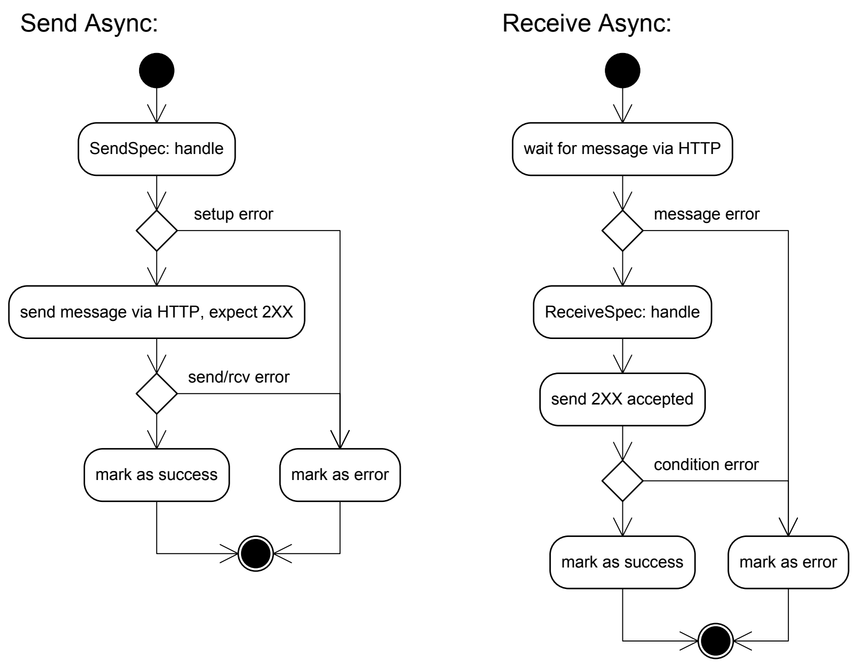


Figure 68: Implementation of SendAsync and ReceiveAsync Activities

The next figure shows the activity diagrams for the send-receive and receive-send synchronous activities.

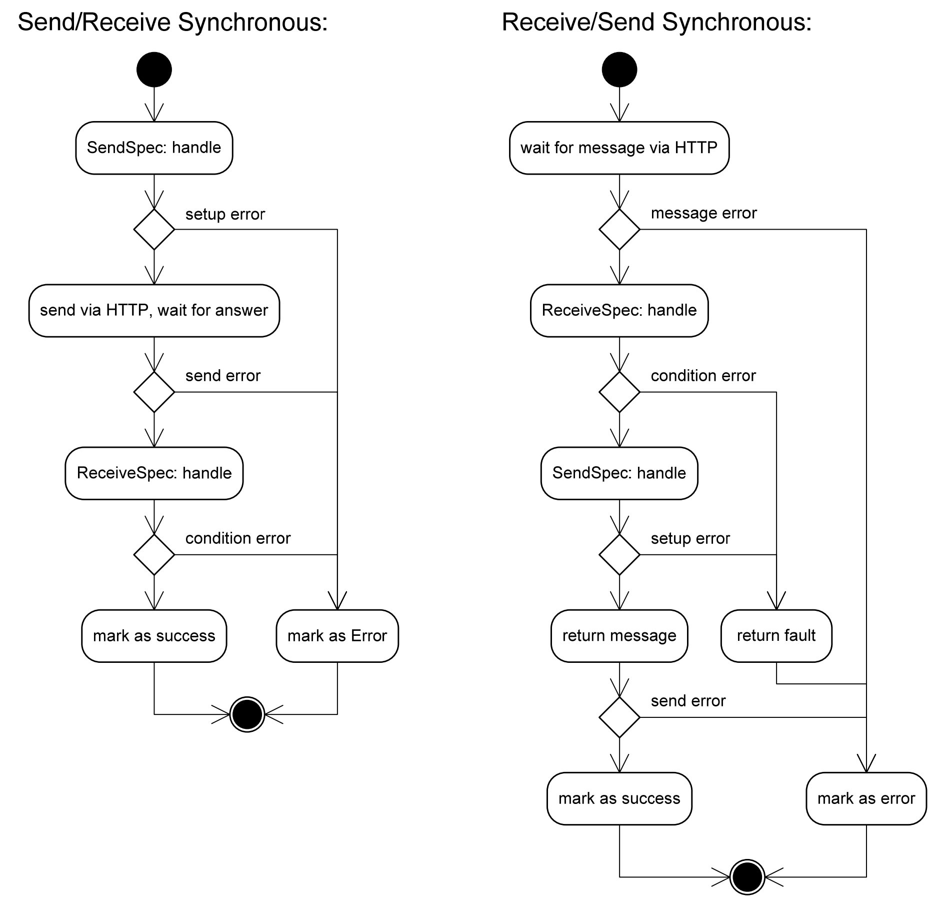


Figure 69: Implementation of SendReceiveSync and ReceiveSendSync Activities

The receive-send synchronous activity on the right-hand side is the only activity within the framework which is able to indicate (technical) faults occurring inside the framework back to the calling

BPEL process, as a synchronous answer may carry a SOAP fault. All other activities employ one-way messaging, except the send-receive synchronous activity, which also may not send a fault upfront.

All four activities displayed so far make extensive use of the receive specification and send specification classes. A data specification encapsulates the complete process of dealing with a SOAP message from the SOAP-encoded form to extracting literal XML to data copying and condition checking.

Figure 70 shows how the preparation of data to be sent and the handling of received data is implemented.

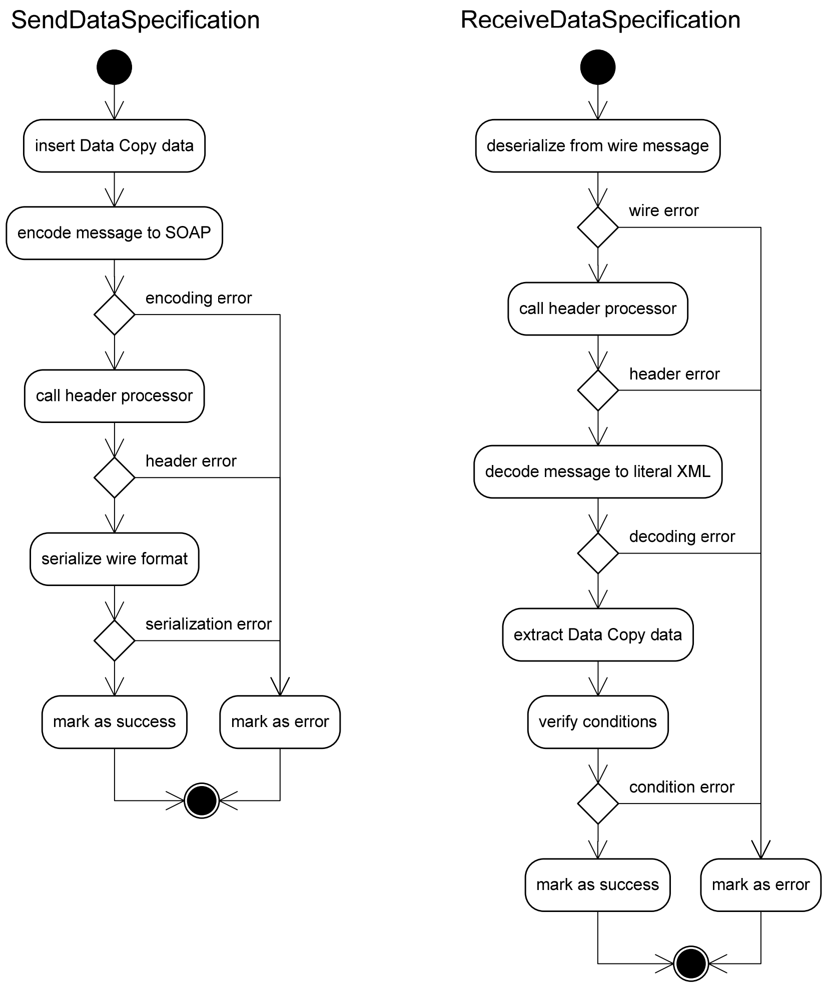


Figure 70: SendDataSpecification and ReceiveDataSpecification

Both diagrams contain links to encoding or decoding a SOAP message. The actual (de)coding process is carried out by a SOAP encoder, which may be plugged into BPELUnit to offer support for a particular SOAP style and encoding; see section 4.2.3 for more details on such encoders.

4.2.2.3 Result storage and error tracking

BPELUnit is a testing framework intended to test a BPEL process for correctness by exchanging messages with it. The purpose of testing is to find errors in the code, and it is thus very important that such errors are detected and stored by the test and later presented to the user.

As pointed out in chapter 3.3.4, there are two general types of faults: application-level *failures*, and middleware-level *errors*. Both are logged by BPELUnit for further analysis.

Contrary to other testing frameworks for conventional programming languages, like for example Java, there is no notion of a "location trace" in BPEL, which could be presented to the user to indicate the source of the problem. Rather, a problem occurs when executing a certain test activity (i.e. while sending a specific message, or receiving a message and testing a condition). BPEL engines adopt different strategies for logging errors; for example:

- ActiveBPEL propagates a fault back to the client (even if the WSDL operation did not specify a fault).
- Oracle does not propagate a fault, but logs it internally and stops processing of the BPEL instance.

In both cases, however, a BPELUnit test will fail sooner or later, at the latest if a timeout occurs because an expected message is not received.

Due to the fact that there is no single source of error information for a failure in the BPEL process, it is important for BPELUnit to log all interactions in all partner tracks and activities to assist the developer in finding a bug. To achieve this, BPELUnit implements a solid chain of information about a test run:

- Each send specification contains the initial literal XML data used, the format of the data after it is formatted as a SOAP envelope, and how it looks on-the-wire (after the header processor added additional information to it).
- Each receive specification contains the incoming plain wire data, the data as a SOAP envelope, the extracted literal data, and all receive conditions and their expected and actual results.

Further up the chain, the activities which use the send and receive specifications add their own data:

- A send activity adds information about the target URL and SOAP HTTP action and about the answer a server provided (only if the answer is unexpected – for example, a 500 error code instead of a 2XX in an asynchronous send. All other cases are already handled by the data specifications).
- A receive activity adds information about the return code of the HTTP operation (if applicable) and any unexpected, non-SOAP payload data.

In case of an error or failure in one partner track, this error of failure is propagated to the test case, which takes on the status of the failure or error. At the same time, the test case runner *aborts* the threads of all other partner tracks, as the test has failed and the test BPEL instance may be discarded.

During a test run and of course also after it has completed, all information discussed above is available via the client API to implementers of clients, who may present them to the user in any way they see fit. This is described in section 4.2.4.

The discussion of the result storage and error tracking mechanisms completes the tour of the framework core. The subsequent chapters will add more detail to the extensions points of BPELUnit, and the API available for writing clients.

4.2.3 BPELUnit core extensions

As mentioned before, the BPELUnit core is a Java component which offers real-life BPEL unit testing by implementing parts of a Web service middleware and by interfacing with BPEL engines for deployment and undeployment of BPEL processes.

To accommodate different engines, SOAP styles, and addressing standards, BPELUnit contains three extension points where it may be extended with code for offering new functionality.

4.2.3.1 BPEL deployers

The first extension point is used to add new BPEL deployers for particular BPEL engines. A deployer consists of the following parts:

- A class implementing the actual deployment code. This class must implement the interface `IBPELDeployer` detailed below.
- A piece of code registering the deployer with a name and the fully qualified class name with the framework. The concrete procedure for this depends on the client implementation, and is detailed in the next section.

- A description of the global options of this deployer. The global options are applicable to the complete BPELUnit installation. They usually contain information about the BPEL server location and credentials for authorization, which are the same for all BPEL test suites.
- A description of the local options for this deployer. The local options are specific for a concrete BPEL test suite and contain information like the name of the PUT BPEL and WSDL file, or the file location of an engine-specific deployment format.

The `IBPELDeployer` interface looks like this:

```
public interface IBPELDeployer {

    public void deploy(String testPath,
                       ProcessUnderTest processUnderTest)
                       throws DeploymentException;

    public void undeploy(String testPath,
                         ProcessUnderTest processUnderTest)
                         throws DeploymentException;

    public void setConfiguration(Map<String, String> options);

}
```

Figure 71: The IBPELDeployer Interface

When deploying a PUT, the framework matches the PUT type given by the user in the test suite document to the code names of registered deployers. When a name matches, the corresponding class is instantiated. Afterwards, the three interface methods are called in the following order:

- The `setConfiguration()` method is used to set the global configuration options, which are specified by the user as part of the BPELUnit installation.
- The `deploy()` method is called in the setup phase of a test suite, instructing the deployer to actually deploy the BPEL process. The method contains two parameters:
  - `testPath`. This is the path in the local file system which points to the root of the test suite document. This path may be needed for resolving file paths of required files, for example, WSDL or XSD files linked from within other files.
  - `processUnderTest`. The process under test object, which contains the name of the PUT, the link to the WSDL file, and the local configuration options given by the user in the test suite specification.
- The `undeploy()` method is the exact opposite of the deploy method, instructing the deployer to undeploy the process from the BPEL server.

BPELUnit already contains three deployers:

- **Oracle BPEL PM Deployer**, implemented by the class `OracleDeployer`, which deploys a PUT to the Oracle BPEL PM server. It is registered with the type `oracle` and may be configured with the following options:
  - Global options: `OracleDirectory` (the directory the Oracle server lives in), `OracleDomain` (the domain of the Oracle server to deploy to), and `OracleDomainPassword` (the password for the domain).
  - Local options: `BPELJARFile` (the path and file name to an Oracle BPEL JAR file to be deployed).
- **ActiveBPEL Deployer**, implemented by the class `ActiveBPELDeployer`, which deploys a PUT to the ActiveBPEL engine. It is registered with the type `activebpel` and may be configured with the following options:

  - Global options: `ActiveBPELDeploymentDirectory` (the directory of the ActiveBPEL deployment directory), and `ActiveBPELDeploymentServiceURL` (the URL of the deployment Web service of ActiveBPEL).
  - Local options: `BPRFile` (the path and file name of an ActiveBPEL BPR BPEL archive to be deployed).
- **Fixed Deployer**, implemented by the class `FixedDeployer`. This deployer basically does nothing. It can be used if the BPEL process has already been deployed to a server by other means and is only invoked by BPELUnit without deployment or undeployment.

4.2.3.2 SOAP encoders/decoders

BPELUnit uses Web service calls to talk to the BPEL process, but allows specification of conditions, data copy operations, and data to be sent in plain, literal XML. To create a Web service call, a SOAP encoder/decoder must convert the literal XML into SOAP XML. The SOAP encoder/decoder extension point can be used to add new SOAP encoders for a certain format (for example, rpc/encoded). A SOAP encoder contains two parts:

- A class implementing the actual encoding/decoding process. This class must implement the interface `ISOAPEncoder`.
- A piece of code registering the SOAP encoder/decoder with a style/encoding and the fully qualified class name with the framework. Again, the concrete procedure for this depends on the client implementation, and is detailed in the next section.

The interface `ISOAPEncoder` looks like this:

```
public interface ISOAPEncoder {

     public SOAPMessage construct(
          SOAPOperationCallIdentifier operation,
          Element literalData)
          throws SOAPEncodingException;

     public Element deconstruct(
          SOAPOperationCallIdentifier operation,
          SOAPMessage message)
          throws SOAPEncodingException;

}
```

Figure 72: The ISOAPEncoder Interface

The first time the framework encounters a SOAP call to be either encoded or decoded with a certain style and encoding, it checks the list of registered SOAP encoders and matches the registered names with the style/encoding of the SOAP call. A document/literal encoder, for example, must thus be registered with the name `document/literal`.

Once instantiated, the encoder instance is used for all subsequent SOAP encoding or decoding processes with the same style and encoding. A SOAP encoder provides two methods:

- The `construct()` method is used to instruct the SOAP encoder to create a complete SOAP envelope out of literal XML data. The method must return a `SOAPMessage`, and takes two parameters:
  - `operation`. This parameter identifies the WSDL options to be used, including the direction of the call (i.e. send, receive, fault).
  - `literalData`. The literal XML data as an XML element node.
- The `deconstruct()` method is used to instruct the SOAP encoder to extract literal data out of a complete SOAP envelope. The method must return the literal XML data as an XML element node and takes two parameters:
  - `operation`. Again, this identifies the WSDL operation.

- `message`. The complete `SOAPMessage` to deconstruct.

BPELUnit already contains two SOAP encoders:

- **document/literal encoder**. The document/literal encoder basically just copies the literal XML data into the payload of the SOAP message.
- **rpc/literal encoder**: The rpc/literal encoder wraps the given literal XML into an outer element in the target namespace of the invoked Web service, thus creating a remote procedure call.

4.2.3.3 Header processors

Asynchronous messaging requires certain fields in the SOAP header to contain message IDs, reply-to addresses, and other information which allows asynchronous callbacks. The header processor extension point can be used to add new callback styles (or any other header mechanism). A header processor consists of three parts:

- A class implementing the actual header processor functionality. This class must implement the interface `IHeaderProcessor`.
- A piece of code registering the header processor with a name and the fully qualified class name with the framework. As mentioned before, the concrete procedure for this depends on the client implementation, and is detailed in the next section.
- A number of configuration options, which the user may use to configure the header processor in a concrete activity in the test suite specification.

The interface `IHeaderProcessor` looks like this:

```
public interface IHeaderProcessor {

     public void processSend(ActivityContext context,
                             SendPackage sendPackage)
                             throws HeaderProcessingException;

     public void processReceive(ActivityContext context,
                                SOAPMessage receivedPackage)
                                throws HeaderProcessingException;

     public void setConfiguration(Map<String, String> options);

}
```

Figure 73: The IHeaderProcessor Interface

A header processor is added to an activity by the user in the test suite specification. When such a declaration is encountered by BPELUnit, the list of registered header processors is checked and the registered names matched with the name given in the activity declaration. The corresponding class is then instantiated. The interface offers three methods:

- The `setConfiguration()` method is used to set the configuration options given by the user in the test suite specification.
- The `processSend()` method is called to process a SOAP envelope before it is sent out to the BPEL process. This may happen as part of a one-way send, a proactive send, or a send as a returning message of a receive-send operation; this is determined by the first parameter.
    - `context`. The activity context provides more information about the activity in which the header processor is called.
    - `sendPackage`. The message to be sent, which includes the SOAP message, the target URL, and the HTTP SOAP action (which may need to be changed by the header processor).

- The `processReceive()` method is called to process a SOAP envelope after it has been received from the BPEL process. This may, again, happen as part of a one-way receive, a proactive receive, or as a receive as part of a send-receive; this is determined by the first parameter.
    - `context`. The activity context provides more information about the activity in which the header processor is called.
    - `receivedPackage`. The received SOAP message.

Note that the calling order of the methods `processSend()` and `processReceive()` depends on the concrete activity. For example, in a send-receive on the client side, the send method is called first, followed by the receive method; the send method must then augment the call with callback information for the BPEL server. In a receive-send on the partner side, the receive method is called first, which must extract the callback information from the envelope and augment the outgoing message with the retrieved information.

BPELUnit contains one header processor without any configuration options and the name `wsa`, which implements the WS-Addressing 1.0 standard.

4.2.4 Clients

The BPELUnit core functionality is offered to users by means of a client, which allows starting and stopping tests, and reviewing the results of a test run. The BPELUnit core provides an API for writing such clients. BPELUnit also contains three clients to start with.

4.2.4.1 Creating a BPELUnit client

Creating a client for BPELUnit involves four tasks:

- Creating a subclass of the abstract class `BPELUnitRunner` and implementing the required methods and functionality. This runner may then be used to start a test suite run.
- Offering a mechanism for registering new extensions, and global configuration for deployers.
- Implementing the `ITestResultListener` interface to be informed about the progress of a test suite run.
- Using the methods of the `ITestArtefact` interface to extract runtime data from test artifacts like tracks, activities, or data specifications.

The `BPELUnitRunner` source code with all relevant methods is displayed in Figure 74. Subclassing `BPELUnitRunner` gives access to three pre-implemented methods (the constructor, and the methods `initialize()` and `loadTestSuite()`. These methods must be used in this order to initialize the runner and load a test suite.

The test suite itself then offers `run()` methods for starting the test, and methods for attaching an `ITestResultListener`.

Subclassing also requires the implementation of the following methods:

- **configure… methods**. These template methods are called from within `initialize()`.
    - `configureInit()` initializes basic BPELUnit values and is called before anything else.
    - `configureLogging()` configures the log4j (Apache 2006b) system and is called second.
    - `configureExtensions()` loads the registered extensions (i.e. BPEL deployers, SOAP encoders and header processors) from wherever the client requires the user to place them (see below for some examples).

  - `configureDeployers()` initializes the deployers with the global options, which are again specified in a predetermined place.

- **create… methods**. These template methods are called by the routines for loading and reading the test suite document, which are triggered by the `loadTestSuite()` method.
  - `createNewDeployer()` returns a new instance of the deployer of the given type.
  - `createNewSOAPEncoder()` returns a new instance of the SOAP encoder of the given SOAP style and encoding in slash-separated format.
  - `createNewHeaderProcessor()` returns a new instance of the header processor of the given name.

```
public abstract class BPELUnitRunner {

     public BPELUnitRunner() { ... }

     public void initialize(Map<String, String> options) { ...  }

     public TestSuite loadTestSuite(File suiteFile) { ... }

     public abstract void configureInit();

     public abstract void configureLogging();

     public abstract void configureExtensions();

     public abstract void configureDeployers();

     public abstract IBPELDeployer createNewDeployer
          (String type);

     public abstract ISOAPEncoder createNewSOAPEncoder
          (String styleEncoding);

     public abstract IHeaderProcessor createNewHeaderProcessor
          (String name);

}
```

Figure 74: BPELUnitRunner Source Code

The `TestSuite` source code contains the following interesting methods:

```
public class TestSuite {

     // Running

     public void setUp() throws DeploymentException { ... }

     public void run() { ... }

     public void shutDown() throws DeploymentException { ... }

     public void abortTest() { ... }

     // Listeners

     public void addResultListener(ITestResultListener listener)
          { ... }

     public void removeResultListener(ITestResultListener listener)
          { ... }
}
```

Figure 75: TestSuite Source Code

As can be seen, the first four methods may be used to control the test cycle. The last two methods can be used to add listeners, which must implement the `ITestResultListener` interface:

```
public interface ITestResultListener {

     public void testCaseStarted(TestCase testCase);

     public void testCaseEnded(TestCase testCase);

     public void progress(ITestArtefact testArtefact);

}
```

Figure 76: ITestResultListener Source Code

The `testCaseStarted()` and `testCaseEnded()` methods report on the test case life cycle. The `progress()` method reports on all artifacts beneath the test cases: partner and client tracks, activities, and data specifications. This method can be used to keep track of current events during the test run, for example for updating a UI.

An `ITestArtefact` offers various possibilities for querying the state of the artifact, as shown in Figure 77:

```
public interface ITestArtefact {

     public String getName();

     public ArtefactStatus getStatus();

     public ITestArtefact getParent();

     public List<ITestArtefact> getChildren();

     public List<StateData> getStateData();

}
```

Figure 77: ITestArtefact Source Code

In particular, the following methods are relevant:

- The `getStatus()` method offers a rich status about the artifact (for example, whether it has passed, failed, has an error, and reasons for an error/failure if applicable).
- The `getStateData()` method returns a list of relevant information about the artifact, for example, SOAP data, wire data, and literal XML data in case of data specifications. For test cases, the state data includes the metadata attached to the test cases.
- The `getParent()` and `getChildren()` methods can be used to display the complete hierarchy in a tree-based view.

BPELUnit comes with three clients: A command line client, an Ant task, and an Eclipse client, which are described shortly in the next sections.

4.2.4.2 The command line client and Ant task

The command line client and the Ant (Apache 2006a) task share a common base implementation, as both are intended to run in a non-graphical shell-based environment, and therefore have the same configuration requirements. Specifically, they employ XML-based configuration files in the BPELUnit installation directory. Both also require a `BPELUNIT_HOME` environment variable to point to the root directory of the BPELUnit installation.

The runners can be called from anywhere in the system. For the *command line client*, a batch file for simpler start up on the Microsoft Windows operating system is provided, but being rather short, it may be easily ported to other environments. The client takes the following options:

```
bpelunit
     [-v]
     [-x=filename]
     [-l=filename]
     testsuite.bpts
     [testcase1] [testcase2] [...]
```

Figure 78: BPELUnit Command Line Client Invocation Options

The command requires the specification of a BPELUnit test suite file, whose default ending is assumed to be `.bpts`. After the file, test case names may be specified, such that only the specified test cases are run. If none are specified, the complete suite is run.

There are three more options:

- `-v` adds detailed, but still readable, output.
- `-x=filename` sends XML output into the given file. The XML output is a fully-fledged (and XML Schema-based) XML result tree of the complete test suite run.
- `-l=filename` sends (extensive) logging messages into the given file.

The *Ant task*, on the other hand, is intended to be called from within Ant scripts. To enable the BPELUnit Ant task, it must first be declared in a `build.xml` file, like this:

```
<typedef name="bpelunit"
         classname="org.bpelunit.framework.ui.ant.BPELUnit">

     <classpath>
          <fileset dir="${BPELUNIT_HOME}/lib" />
     </classpath>

</typedef>
```

Figure 79: BPELUnit Ant Task Type Definition

Once defined, the task may be used like this:

```
<bpelunit testsuite="somedir/someSuite.bpts"
          bpelunitdir="${BPELUNIT_HOME}">

     <output style="XML" file="xmloutput.xml" />
     <logging level="DEBUG" file="logoutput.txt" />
</bpelunit>
```

Figure 80: BPELUnit Ant Task Invocation

The task requires the specification of a test suite file; the BPELUnit home directory may either be specified here or using the `BPELUNIT_HOME` environment variable. The task may additionally carry any number of `output` and `logging` statements, each with or without the file attribute.

- If the file attribute is specified, the output is sent to the given file.
- If it is not specified, the output is sent to the console.

In case of the `output` statement, the `style` attribute may carry either `PLAIN` or `XML`. `PLAIN` creates a human-readable trace of the run; `XML` creates a fully-fledged XML test result tree.

In the case of the `logging` statement, the `level` attribute specifies the log4j level (Apache 2006b) for the logging output. In most cases, the `INFO` level will be sufficient.

Both the command line client and the Ant task require two configuration files in the BPELUnit installation `conf/` directory: `extensions.xml`, and `configuration.xml`. These two files are used to add new extensions (i.e. BPEL deployers, SOAP encoders/decoders and header processors) to the system, and to configure the deployers with global options.

The extensions are specified in the first XML file (Figure 81). Three examples are given; one for each extension point: one deployer (`activebpel`), one SOAP encoder (`document/literal`), and one header processor (`wsa`).

```
<extensionRegistry>

     <deployer
          type="activebpel"
          extensionClass="org.bpelunit...ActiveBPELDeployer"
          />

     <encoder
          type="document/literal"
          extensionClass="org.bpelunit...DocumentLiteralEncoder" />

     <headerProcessor
          type="wsa"
          extensionClass="org.bpelunit...WSAHeaderProcessor" />

     ...

</extensionRegistry>
```

Figure 81: Extension Registry of Command Line Client and Ant Task

Of course, the referenced classes and their prerequisites must be on the `CLASSPATH` when BPELUnit is run; the easiest way of achieving this is placing the JARs into the BPELUnit `lib/` folder.

Figure 82 gives an example of the configuration file `configuration.xml`, which contains the global options for registered deployers. As an example, the configuration for the Oracle deployer is displayed.

```
<testConfiguration>

     <configuration deployer="oracle">
          <property name="OracleDirectory">...</property>
          <property name="OracleDomain">...</property>
          <property name="OracleDomainPassword">...</property>
     </configuration>

     ...

</testConfiguration>
```

Figure 82: Configuration of Deployers in Command Line Client and Ant Task

4.2.4.3 The Eclipse client

The BPELUnit Eclipse (Eclipse 2006) client offers BPELUnit functionality within the Eclipse IDE. Its invocation and functionality is similar to the Eclipse JUnit integration.

Being an Eclipse plug-in, the client has access to the usual plug-in mechanisms of Eclipse and therefore also uses them to allow addition of new extensions (deployers, SOAP encoders, and header processors) and for global configuration of deployers. The client also offers a basic XML editor for the test suite (a graphical editor is added by the tool support, see next section).

The Eclipse client is also described thoroughly in the Eclipse online help system, which is available after plug-in installation.

Running a test suite from within Eclipse with the BPELUnit client installed works like running a Java application:

- By right-clicking the test suite file in the resource or package explorers, and selecting *Run As > BPELUnit Test Suite*.
- Via the main menu (*Run > Run…*).

The latter is displayed in Figure 83.

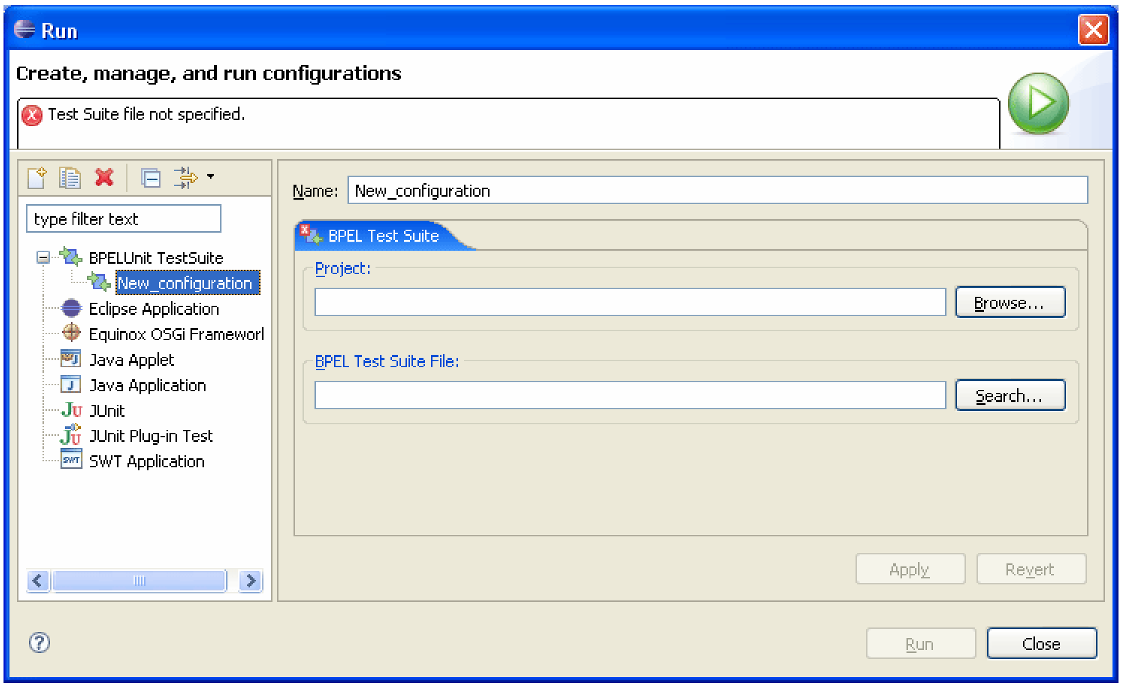
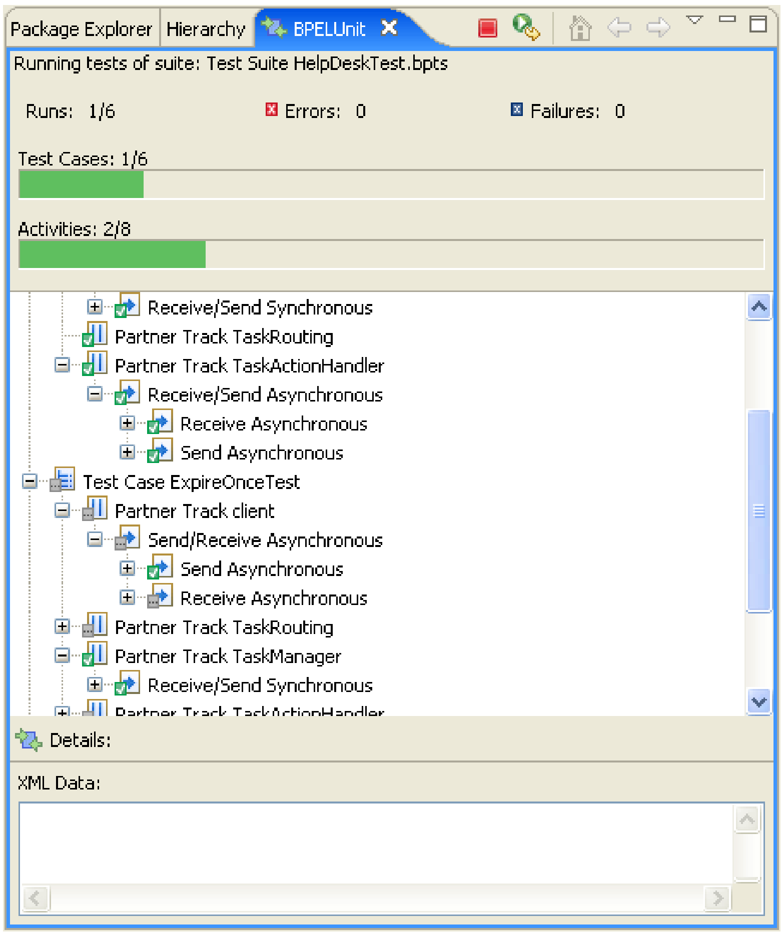


Figure 83: BPELUnit Launch Dialog in Eclipse

After a test suite has been launched, the *result view* comes up, which displays information about the current test run. The view is updated as the test progresses. Depending on the current run, the view will look something like Figure 84:



Running tests of suite: Test Suite HelpDeskTest.bpts

Runs: 1/6 Errors: 0 Failures: 0

Test Cases: 1/6

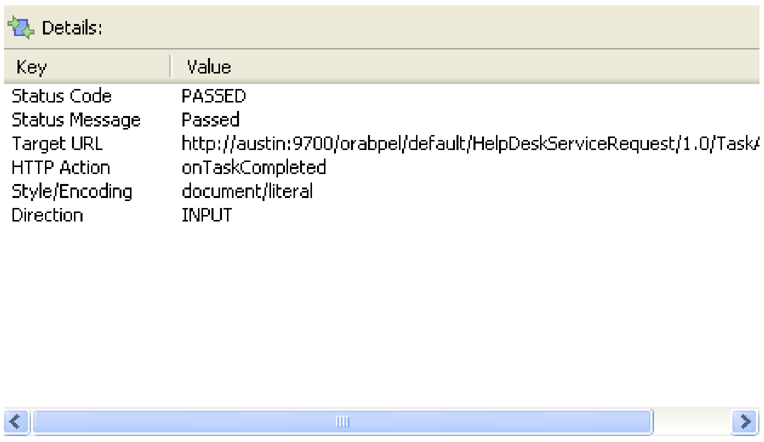
Activities: 2/8

Figure 84: Running a Test in the BPELUnit Eclipse Runner

The view contains two JUnit-like bars. The first bar shows the test case progress; the second bar shows the activity progress inside the current test case. As expected, the bars turn red and stay red if any failure or error occurs.

The tree view immediately below the bars displays the structure of the current test suite. Each top-level node represents a test case, which contains partners, which in turn contain activities. Initially, all nodes are displayed in grey, as they have not been executed yet. When executed, they will turn green in case of a successful execution, or red in case of a failure or error.

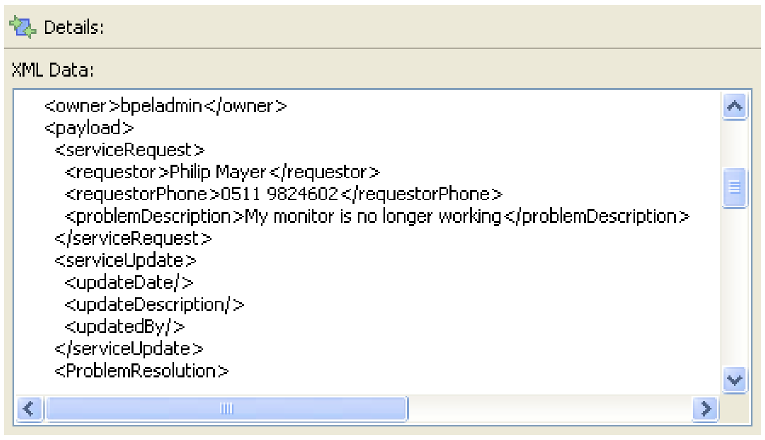
Each node may be selected to display more information about the node in the detail pane at the bottom of the view. For example, selecting a send activity will yield information on where the data was sent:



Key	Value
Status Code	PASSED
Status Message	Passed
Target URL	http://austin:9700/orabpel/default/HelpDeskServiceRequest/1.0/Task/
HTTP Action	onTaskCompleted
Style/Encoding	document/literal
Direction	INPUT

Figure 85: More information about a Send Activity

Selecting an XML node beneath a send or receive activity displays the actual data that was sent:



XML Data:

```
<owner>bpeladmin</owner>
<payload>
 <serviceRequest>
  <requestor>Philip Mayer</requestor>
  <requestorPhone>0511 9824602</requestorPhone>
  <problemDescription>My monitor is no longer working</problemDescription>
 </serviceRequest>
 <serviceUpdate>
  <updateDate/>
  <updateDescription/>
  <updatedBy/>
 </serviceUpdate>
 <ProblemResolution>
```

Figure 86: XML Data Display

The BPELUnit result view also allows users to stop the test at any time (red stop button at the top), and re-launch the test (green/yellow re-run button at the top).

As mentioned above, the BPELUnit Eclipse client offers Eclipse-based extension points for adding new deployers, SOAP encoders, and header processors. The process for adding new Eclipse extensions is described in the Eclipse documentation.

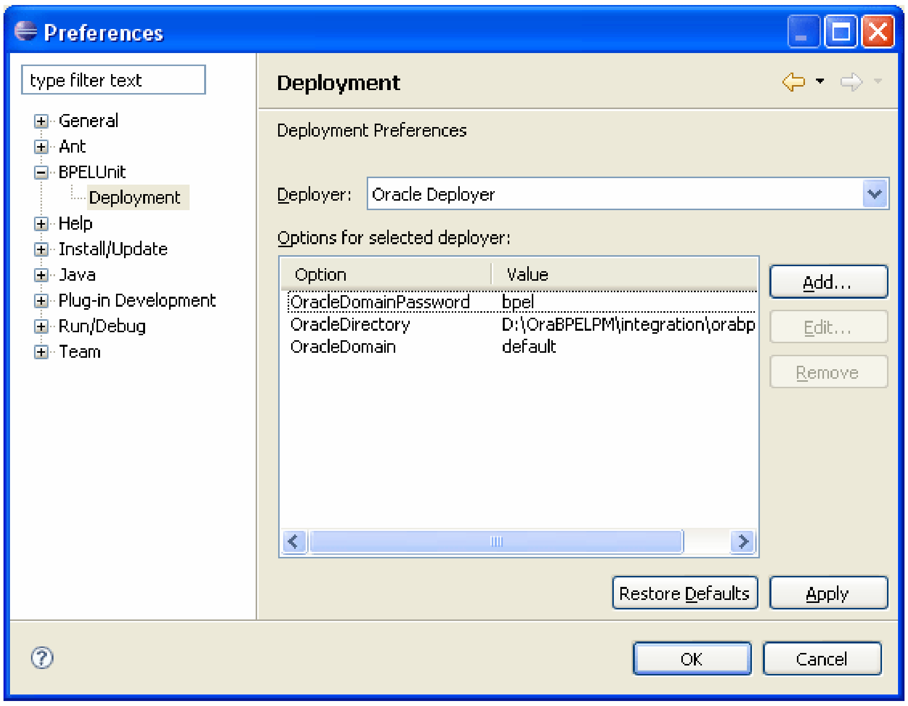
As an example, Figure 87 shows the `plugin.xml` declarations needed for adding new extensions, one for each kind.

```
<extension point="org.bpelunit.framework.client.eclipse.bpelDeployer">
     <bpelDeployer
          id="oracle"
          name="Oracle Deployer"
          deployerClass="org.bpelunit...OracleDeployer"
          general_options="..."
          suite_options="..."
     />
</extension>

<extension point="org.bpelunit.framework.client.eclipse.soapEncoder">
     <soapProcessor
          id="document/literal"
          name="Document/Literal Encoder"
          encoderClass="org.bpelunit...DocumentLiteralEncoder"
     />
</extension>

<extension point="org.bpelunit.framework...eclipse.headerProcessor">
     <headerProcessor
          id="wsa"
          name="WS-Adressing Header Processor"
          processorClass="org.bpelunit...WSAHeaderProcessor"
     />
</extension>
```

Figure 87: Defining New Extensions for the BPELUnit Eclipse Client

Finally, the BPELUnit client also offers a configuration panel for adding or changing the global options of deployers. This panel can be reached via the Eclipse preferences dialog, and is displayed in the next figure.

The configuration options are stored with the workspace.



Deployer: Oracle Deployer

Options for selected deployer:

Option	Value
OracleDomainPassword	bpel
OracleDirectory	D:\OraBPELPM\integration\orabp
OracleDomain	default

Figure 88: Changing Global Deployment Options in Eclipse

With the description of the Eclipse client, the BPELUnit framework has been described in all of its aspects. The next section will introduce tool support for writing the test suite specification files.

4.3 BPELUnit tool support

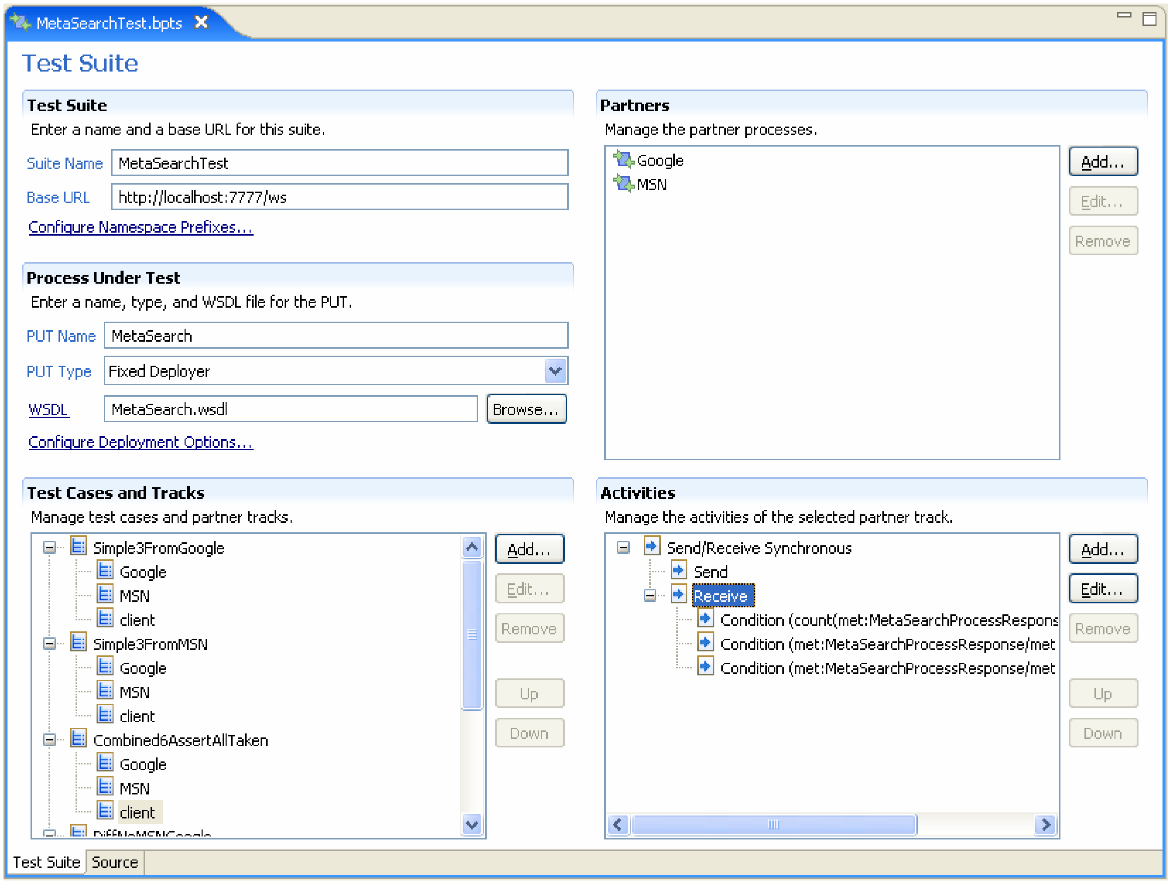
Like the BPEL programming language itself, the BPELUnit test suite specification language is XML-based, allowing for easy integration of artifacts, especially literal XML data to be sent to the PUT. However, as in BPEL, the XML language is error-prone to write by hand, which is the reason why many vendors of BPEL editors have chosen to implement graphical editors for BPEL.

For the same reason, BPELUnit comes with a graphical editor for the test suite specification (`.bpts`) files. The editor is implemented as a set of Eclipse plug-ins, and offers a complete editing environment for all aspects of the test specification.

The editor can be invoked by opening `.bpts` files placed in any Eclipse project. By selecting *File > New… > Other…* and then *BPELUnit > BPELUnit Test Suite*, a new `.bpts` file with a basic test outline is created and opened for editing.

4.3.1 Editor overview

The BPELUnit test suite editor offers a graphical interface for editing all parts of a test suite document. The editor has two pages; a *visual page* which allows editing by means of a graphical user interface and wizards, and a *source page* which allows editing of the test suite source code.


MetaSearchTest.bpts

Test Suite

Test Suite
Enter a name and a base URL for this suite.
Suite Name: MetaSearchTest
Base URL: http://localhost:7777/ws
Configure Namespace Prefixes…

Process Under Test
Enter a name, type, and WSDL file for the PUT.
PUT Name: MetaSearch
PUT Type: Fixed Deployer
WSDL: MetaSearch.wsdl
Configure Deployment Options…

Test Cases and Tracks
Manage test cases and partner tracks.
- Simple3FromGoogle
  - Google
  - MSN
  - client
- Simple3FromMSN
  - Google
  - MSN
  - client
- Combined6AssertAllTaken
  - Google
  - MSN
  - client

Partners
Manage the partner processes.
- Google
- MSN

Activities
Manage the activities of the selected partner track.
- Send/Receive Synchronous
  - Send
  - Receive
    - Condition (count(met:MetaSearchProcessRespons
    - Condition (met:MetaSearchProcessResponse/met
    - Condition (met:MetaSearchProcessResponse/met

Figure 89: Test Suite Editor Visual View

The visual editor window consists of five sections:

- **Test Suite section.** This section shows the basic options of the test suite, like its name, and base URL, and contains a link to the namespace prefix editor.
- **Process Under Test section.** This section offers configuration of the process under test (PUT): its name, type, WSDL file, and deployment options.
- **Partners section.** This section allows adding, editing, and removing of partners.

- **Test Cases and Tracks section**. This section allows adding, editing, removing, and moving of test cases, and their partner tracks.
- **Activities section**. This section allows adding, editing, removing, and moving activities of a selected partner track.

Changes made in either the visual or the source page are immediately reflected in the other page, respectively. It is thus safe to switch between the pages at any given time.

4.3.2 Configuring the test suite

The first three sections – test suite section, PUT section, and partners section – correspond to the `deployment` part of the test suite document. As expected, all attributes and elements of the deployment section can be changed, ranging from the name of the test suite via PUT name, type, and deployment options, to the partners of the PUT and their WSDL files.

Additionally, there is a link to the namespace editor. Namespaces are very important in the test suite document, as the literal data to be sent to the PUT and the XPath conditions used for verifying incoming transmissions and copying values must be formulated with the right namespace URIs.

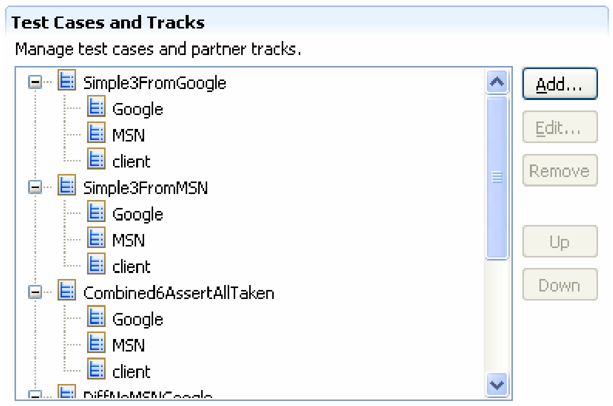
In XML documents, namespace declarations may be added to any element and used in all sub elements of that element. Of course, this also works for test suite documents. However, to ease the handling of XML namespaces in test suite specification files, the graphical editor places all namespace declarations in the root element (`testSuite` element), and offers an editor for these namespaces.

When adding WSDL files to a test suite – either for the PUT or a partner – their target namespaces are automatically added and a prefix is generated. These can later be changed in the editor.

4.3.3 Editing test cases and tracks

Once the PUT and all partners have been configured, test cases may be added in the test case section. To recall, each test case contains one client track and (possibly) several partner tracks; these tracks each consist of a sequence of activities which describe the frameworks' actions on behalf of the partner in the given test case.

Each test case is identified by a human-readable name. Test cases are displayed as root nodes in the tree view. A test case may have any number of partners, selected from the partners in the partners section, and one client, which are displayed as children of the test cases in the tree.


Test Cases and Tracks
Manage test cases and partner tracks.

- Simple3FromGoogle
  - Google
  - MSN
  - client
- Simple3FromMSN
  - Google
  - MSN
  - client
- Combined6AssertAllTaken
  - Google
  - MSN
  - client

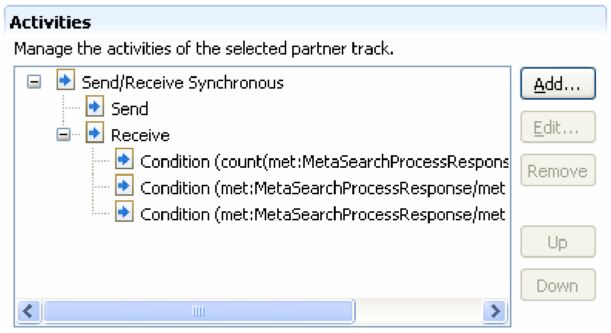
Figure 90: Test Cases and Tracks Section

For the users' convenience, a new test case is always generated with all available partners and the client as partner tracks. However, by right-clicking on a test case, new partner tracks may be added; by selecting a partner track, it can be edited or removed. Note that it is not possible to remove the client track, as each test case needs a client. The track may remain empty, though, enabling inheritance of client activities.

Selecting a track enables the activity section on the right, which allows the user to edit the activities of the selected track.

4.3.4 Editing activities

The activities section contains a tree view. The root nodes in the tree represent activities; their children are subactivities and/or special nodes like conditions, copy operations, or header processors.


Activities

Manage the activities of the selected partner track.

```
Send/Receive Synchronous
    Send
    Receive
        Condition (count(met:MetaSearchProcessRespons
        Condition (met:MetaSearchProcessResponse/met
        Condition (met:MetaSearchProcessResponse/met
```

Figure 91: Activity Section

As discussed in previous chapters, BPELUnit supports six types of activities, which may be added by right-clicking in the tree and selecting *New...*, or by clicking the *Add…* button.

Each unique send or receive requires a WSDL operation. In case of send-only, receive-only, and synchronous two-way activities, exactly one operation must be specified. The asynchronous two-way activities require two operations.

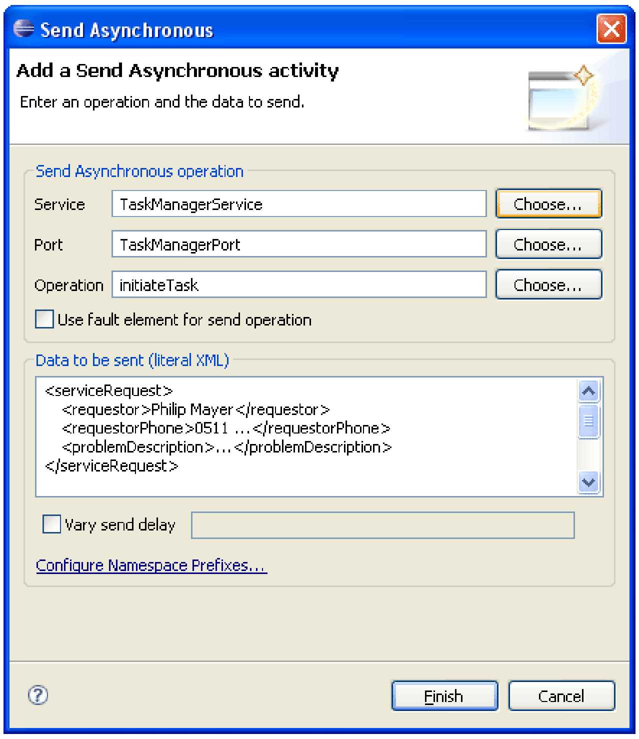
Besides a send and/or receive specification, each activity may contain the following components:

- **Copy operations**, which copy data from in- to output.
- A **header processor**, which handles callback addressing.

The tool support provides a customized wizard for each activity. The send and receive, copy operation, and header processor input fields are available in the activity-specific wizards as needed by the activity type.

4.3.4.1 Send specification

As an example, the send specification of a send-only activity looks like this:


Send Asynchronous

Add a Send Asynchronous activity

Enter an operation and the data to send.

Send Asynchronous operation

Service: TaskManagerService

Port: TaskManagerPort

Operation: initiateTask

☐ Use fault element for send operation

Data to be sent (literal XML)

```
<serviceRequest>
  <requestor>Philip Mayer</requestor>
  <requestorPhone>0511 ...</requestorPhone>
  <problemDescription>...</problemDescription>
</serviceRequest>
```

☐ Vary send delay

Figure 92: A Send-Only Activity

The send specification requires a *service*, *port*, and *operation* to be specified. By selecting the *Choose...* buttons, a service, port and operation may be selected from the WSDL file of the partner of the selected track.

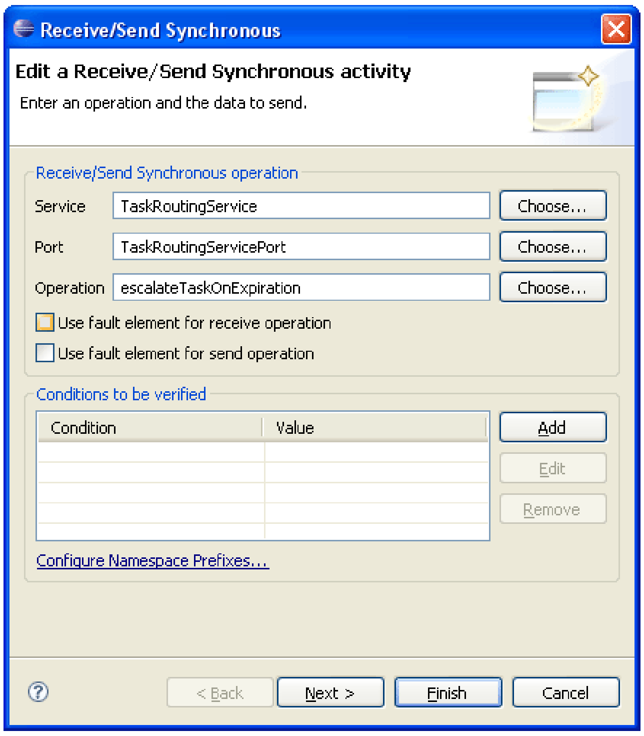
To send a SOAP fault, the *Use fault element for send operation* check box may be selected.

The data to be sent is entered in the *literal XML field*. The XML will be validated as it is typed in, informing the user of any syntactical error. As mentioned before, namespaces are managed globally by the tool support; thus, namespace declarations need not be included in the literal XML. To change namespaces on-the-fly a link to the namespace editor is provided.

As already mentioned, each send operation may also contain a send delay sequence – a comma-separated list of delay times in seconds.

4.3.4.2 Receive specification

The following screenshot shows a receive specification inside of a receive-send synchronous activity wizard:


Receive/Send Synchronous
Edit a Receive/Send Synchronous activity
Enter an operation and the data to send.
Receive/Send Synchronous operation
Choose...
Choose...
Choose...
Use fault element for receive operation
Use fault element for send operation
Conditions to be verified
Condition
Value
Add
Edit
Remove
Configure Namespace Prefixes...
< Back
Next >
Finish
Cancel

Service TaskRoutingService
Port TaskRoutingServicePort
Operation escalateTaskOnExpiration

Figure 93: A Receive-Send Synchronous Activity

Being a synchronous activity, the receive and send specifications share one WSDL operation (at the top) – therefore, the operation block allows fault specification for both send and receive.

A receive specification consists of *conditions* which are checked against the incoming XML data. The *Add*, *Edit*, and *Remove* buttons may be used to manage these conditions.

4.3.4.3 Advanced options

Besides the send and receive specifications, the activity wizards also contain pages for editing data copy operations and header processors. These pages allow adding, editing, and removing of copy operations, and selecting a header processor from a list of all registered processors and adding, editing, and removing header processor options.

As all UI elements discussed previously, these advanced options are also described thoroughly in the Eclipse online help system, which is available after plug-in installation.

4.4 Summary

In this chapter, the design and implementation of the BPELUnit testing framework have been presented. BPELUnit aims at being a member of the xUnit family, extending its reach to the BPEL programming language.

BPELUnit contains several components:

- The **test specification language**, which is an XML-based language for describing BPEL test suites and test cases.
- The **framework core**, which is a Java component implementing the actual test logic.
- **BPELUnit clients**, which use the framework API and allow users to run test suites.
- The **BPELUnit tool support**, which offers a convenient graphical user front-end to the test specification files.

The BPELUnit framework, its clients and tool support have already been evaluated in practice by testing several BPEL processes. The next chapter presents two examples.

5 Example processes

This chapter presents two example BPEL processes and their test suites. The first was written from scratch as an example of a more technical workflow (Meta Search); the second was written by Oracle as a demonstration of their BPEL server (Help Desk Service).

5.1 The Meta Search process

The Meta Search process is a BPEL process which implements a search ranging over multiple Internet search engines. In particular, the Meta Search process relays a query to the search engines Google and MSN, combining the results from both engines, eliminating duplicates, and returning the result to the client.

Both Google and MSN have Web services available for performing a search which are used in the BPEL process. Both use WSDL for service description and SOAP for messaging. MSN uses document/literal style, whereas Google uses rpc/encoded style. As has been mentioned in the previous chapter, BPELUnit does not support the rpc/encoded style; thus, a bridge BPEL process was created to translate the requests from document/literal to rpc/encoded. However, this does not affect the overall logic, as the bridge offers the same operations as the original Web service, only with a different encoding.

The Meta Search process thus has one client and two partners – the MSN service from Microsoft, and the GoogleBridge BPEL process, which in turn calls the Google service from Google. The interface for the client has one synchronous operation with the following parameters:

- **Input message**:
  - **Query (String)** – the search query.
  - **Language (String)** – the requested language of the search results (ISO 639-1, alpha-2 code), for example "de" or "en".
  - **Country (String)** – the requested originating country of the search results (ISO 3166-1, alpha-2 code), for example "DE" or "US".
  - **MaxResults (int)** – the maximum number of results to return.
- **Output message**:
  - **NoResults (int)** – the result count.
  - **List of Results**, each containing:
    - **URL (String)** – the URL.
    - **Title (String)** – the title of the page.
    - **Snippet (String)** – a short summary of the page.
    - **From (String)** – indicates the engine which found this page.

The overall flow of the implementing BPEL process consists of the following steps (see Figure 94):

- **receiveInput**: At the beginning, a search query is received by the BPEL process.
- **GoogleSearchScope** and **MSNSearchScope**. These two scopes run in parallel inside a `flow` activity. Each scope invokes the corresponding search engine and waits for the response. If none is received or an error occurs, the result from this engine is ignored.
- **Switch**: If some results were received at all, the **IterationScope** is started; otherwise, a null output is generated.
- **IterationScope**. This scope contains the logic for merging the results, and eliminating duplicates.
- **replyOutput**: The result is returned to the client.

Figure 94 shows a simplified diagram of the Meta Search BPEL process, taken from the Oracle JDeveloper editor. The complete process contains some 70+ BPEL activities:

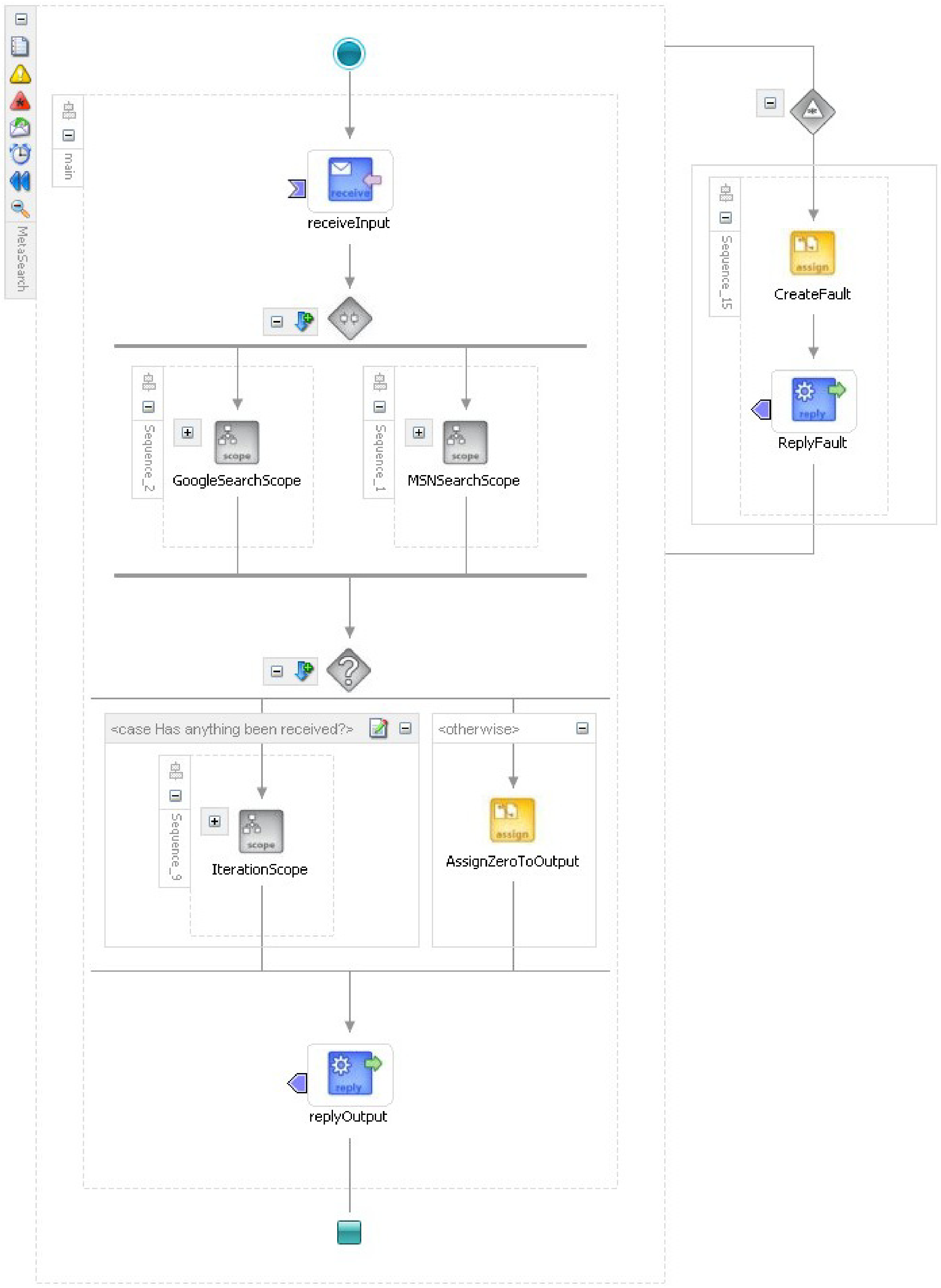


Figure 94: The MetaSearch Process (abbreviated)

The BPEL processes uses one operation from each partner – the `doGoogleSearch` operation from GoogleBridge, and the `Search` operation from MSN. Each of these operations takes roughly the same input parameters as the BPEL process itself. Additionally, an identification string (a kind of license) is required, which must be requested from Google or Microsoft.

Each of the Web services returns a list of results in a slightly different format (different XML Schema, different names, etc.); the BPEL process combines them in the **IterationScope**.

A unit test of the Meta Search process must form a harness around the process, thus simulating the client and the GoogleBridge and MSN Web services. Each test case thus contains one client track and two partner tracks. The client track contains one send-receive synchronous activity, whereas the partner tracks each contain a receive-send synchronous activity.

The following five test cases have been used to find bugs in the code; each is taken from a different test equivalence class.

- **Google only**. This test case only sends results from Google, but not from MSN. The BPEL process should not fail in this case, but return only the Google results without an error.
- **MSN only**. The same for MSN.
- **Combined distinct results, equal length**. This test case sends the same number of results from both Google and MSN; all results must be included. This test case is varying.
- **Combined distinct results, different length**. This test case sends results from both Google and MSN, but in different numbers; all results must be included.
- **Maximum results**. This test case asserts that the maximum number of results is always respected.
- **Overlapping results**. This test case sends some same URLs from both engines; those must be filtered by the BPEL process.

In the case of the meta search process, unit testing greatly helped in development of the process, as it was possible to simulate different behavior of the search engines Google and MSN to test different possible scenarios. Development of the meta search process was done iteratively, interleaving development and testing. Several bugs have been found with the help of the unit tests:

- **Iteration length of loop for checking duplicates**. The Oracle BPEL server pre-creates the structure of XML variables, such that one result element (a search engine hit) for the client is already in-place when adding the ones received from Google and MSN. This led to an iteration count which always skipped the last element, missing some duplicates.
- **Result fault**. During development, the final BPEL reply activity in the process was specified with a fault name. As a result, the (normal) content was incorrectly returned as a SOAP fault.
- **Array length**. In BPEL/XML, an array starts at one (1), not zero (0). Therefore, the condition for the end of an array is `<= max`, not `< max`, as in Java.

As can be seen from these examples, the bugs found were typical programming errors, which can be spotted by testing examples from all test equivalence classes.

5.2 The Help Desk Service process

The second process tested was the Help Desk Service process, which is included as part of the Oracle samples for the Oracle BPEL PM server.

This process implements a typical help desk scenario: An employee has a problem and requests help – in form of a Web service call – from the department responsible. The help request is sent to a reviewer, who can either resolve the problem himself, or delegate the request to another reviewer. If the request is not handled within an appropriate time, it is escalated up a predefined employee hierarchy. Finally, it is returned to the client with either a RESOLVED status (containing information on how to resolve the problem) or an UNRESOLVED status (problem could not be resolved).

The interface for the client has one asynchronous operation with the following parameters:

- **Input message**:
  - **requestor (String)**. Name of the person who makes this request.
  - **requestorPhone (String)**. Phone number of the requestor.
  - **problemDescription (String)**. A description of the problem.

- **Output message** (additionally, input elements are repeated here):
    - **updateDate (Date)**, **updateDescription (String)**, and **updatedBy (String)**: Meta information about the help desk service request.
    - **problemResolution (String)**. A description of how the problem can be resolved.
    - **problemResolvedBy (String)**. Name of the employee who resolved the problem.

The BPEL process calls up to three partners, which implement the reviews and talk to the employees involved. All three are provided with WSDL descriptions and use a document/literal SOAP encoding.

- **TaskManager**. The task manager initializes a task with pre-defined default values. The call is synchronous.
- **TaskActionHandler**. The task action handler is called to request an answer for a service request from an employee. The call is asynchronous.
- **TaskRouting**. The task routing service is called during escalation, and targets the request to another employee. The call is synchronous.

All three services are provided by Oracle, but can – of course – be simulated by the BPELUnit test framework in a test suite.

The main parts of the implementing BPEL process all deal with a so-called *task*, which is created from the request after it is received, updated during the lifetime of the BPEL process instance, and then (in part) copied into the result message for the client. All three partners expect such a task as an input to their operations, and return an updated task.

The overall flow of the BPEL process consists of the following steps (see also Figure 95):

- **receiveInput**: First, a help desk request is received from the client.
- **initiateTask**: A task is generated for the help desk request by invoking the task manager Web service. The task is given some meta data, like the first reviewer. The state of the task is set to ASSIGNED.
- A **loop** follows, which has two internal operations. The loop terminates if the task has the status RESOLVED or UNRESOLVED.
    - **invokeActionHandling**. The task is turned over to the task action handler, which calls upon a human to deal with the task. If the action handler is not able to reach the human or if a failure occurs, it returns with an expiration error (task is set to EXPIRED). The employee called may assign any state or other meta information to the task.
    - **checkExpirationAndEscalate**. The task is checked: If it is expired, it is turned over to the task routing service, which re-targets the task, i.e., sets a new reviewer based on the company hierarchy.
- **readUpdatedTask**. The updated task is prepared for returning it to the user.
- **createAndSendResult**. A result for the original requestor is created from the task, and sent back to the client.

Figure 95 gives an abbreviated overview of the Oracle Help Desk Service. The actual BPEL process contains some 40+ activities.

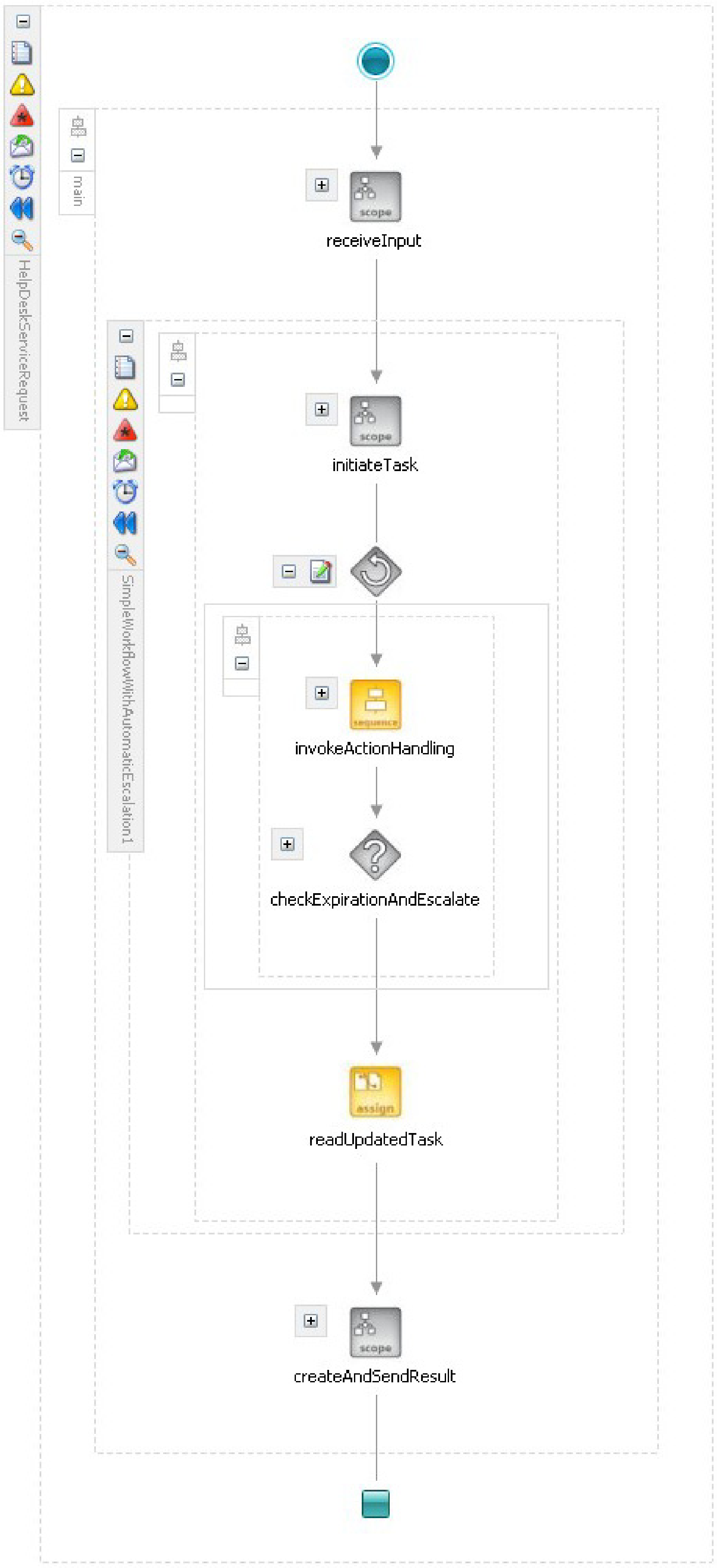


Figure 95: The Help Desk Service (abbreviated)

To test this BPEL process, six test cases were created:

- **Non-escalation test**. The first test case simulates a simple help desk request: The first reviewer already knows the answer and provides it to the client.
- **Delegate test**. The second test case simulates a re-assignment: The first reviewer reads the request, but does not know the answer, and delegates the request to another reviewer.
- **One expiration and escalation test.** This test case simulates a timeout – in those cases, the assigned reviewer did not even read the request, and it is automatically re-assigned (escalated to the next level).
- **Initiation failure**. This test case tests failures in the creation of the task.
- **Unresolved test**. This test simulates an unresolved help desk request, which must also be returned to the client.
- **Fail in expire test**. This test case simulates a failure in the expiration process, i.e., no further escalation level is possible, or something went wrong during re-assignment.

The test cases in this example use inheritance to factor out common tasks, like the initial request from the client.

In the Help Desk Service process example, unit testing greatly helped in understanding the already existing BPEL process as it was easily possible to test different behaviors of the process by sending in different data and examining the results. No bugs were encountered during testing.

5.3 Summary

The two example BPEL processes presented in this chapter serve very different purposes, and their implementations reflect this fact:

- The **Meta Search process** is an example of a technical workflow. Although using two partners to achieve its goal, the process also contains a good deal of business logic itself. While developing this service, unit testing was used successfully to thoroughly test the implemented logic by using specially prepared data to test corner cases, thereby finding and fixing bugs in the implementation.
- The **Help Desk Service process**, on the other hand, is an example of a delegating workflow. It uses three partners to achieve its goal, which implement most of the business logic themselves, only leaving the delegating logic for the BPEL process. With fewer activities dedicated to actual business logic, there is also less room for mistakes, and in fact, no bugs were found when testing this process. However, unit testing was employed successfully to study the behavior of the process and test error conditions which cannot easily be achieved during normal operation (for example, by simulating SOAP faults).

Thus, both processes have been successfully tested with BPELUnit, establishing trust in the functionality and tolerance for faults of the implemented business logic.

6 Conclusion and outlook

This chapter summarizes the contributions of this thesis, provides an overview of work still to be done in the area of BPEL composition testing, and concludes the thesis.

6.1 Summary

The main goal of this thesis was the creation of a framework for testing BPEL compositions. This goal has been reached by:

- Discussion of the realm of BPEL composition testing, defining the basic test approach, the unit under test, and unit test cases.
- Definition of a generic BPEL unit testing architecture, which can be used to create concrete BPEL unit testing frameworks.
- Design and implementation of the concrete testing framework BPELUnit and additional tools, enabling developers to test BPEL compositions in a real-life environment in a simple and easy way.

At first, the basic testing approach was defined as white-box unit testing targeted at the individual BPEL developer. The unit under test was defined as the BPEL process itself; and the unit test cases as a harness around the process under test, receiving data from and sending data to all of its interfaces.

Building on this, a generic, four-layer BPEL unit testing architecture was discussed. The four layers (specification, organization, execution, and results) were described thoroughly, and alternate realization possibilities presented. In the following, the concrete BPEL unit testing framework *BPELUnit* was derived from this architecture.

The technical design and implementation of BPELUnit were discussed next, ranging from the test specification XML format itself to the core implementation, and finally extending to clients and extension points. Additionally, tool support for writing the test specification documents was presented.

Finally, two example processes and their BPEL test cases were described, which may serve as a starting point for further evaluation of BPELUnit.

6.2 Future work

BPELUnit is a complete implementation of a BPEL unit testing framework which can be used as-is to test BPEL processes. It offers various extension points, which can be used to add the following components:

- **SOAP encodings**. Right now, BPELUnit contains encoders for the rpc/literal and document/literal styles. The extension points may be used to add new encodings.
- **Callback addressing**. BPELUnit contains a header processor implementing the WS-Addressing specification; others may be needed.
- **BPEL deployment**. Currently, BPELUnit contains two deployers, one for the Oracle BPEL PM server, and one for the ActiveBPEL engine. Other engines may be added.
- **Clients**. BPELUnit comes with three default clients – a command line client, an Ant task, and a JUnit-like Eclipse test runner. Further clients may be implemented on top of the existing test runner API.

Besides extending BPELUnit, the following research topics are interesting and will provide more insight into the area of BPEL composition testing:

- **Gathering data about BPEL usage**. The BPEL language is quite new, and up until now, no reports of successful or unsuccessful large-scale BPEL deployments are available. To gain more insight into common BPEL pitfalls and deployment issues, a survey might be conducted to gather such data. An examination of existing BPEL processes might lead to design patterns, or anti patterns, for BPEL.

- **Deployment descriptor analysis**. As pointed out in chapter 2.3, BPEL deployment and deployment descriptors are highly vendor-specific and currently non-interchangeable. An analysis of deployment approaches could provide more insight into the different mechanisms and may offer a chance for a standardization of such features.
- **Debug API**. BPEL engines currently available implement different APIs for the simulation – or debugging – of BPEL processes. As in deployment descriptors, a common debug API – as, for example, exists for Java virtual machines, could offer new possibilities for BPEL composition unit testing.
- **Metric support**. An interesting question is how to calculate metrics, like for example code/activity coverage or completed paths.
- **Applying the test framework to different languages**. The generic test framework presented in this thesis makes only few assumptions about the actual composition language. Basically, it can be used for any Web service-based composition language whose programs talk to other Web services. It will be interesting to see the framework applied to other languages.

6.3 Conclusion

As has been pointed out in chapter 3.4, the BPELUnit framework breaks new ground by providing a complete solution for the specification and execution of white-box BPEL unit tests and for gathering and presenting the results, enabling the test of both synchronous and asynchronous operations with any number of partners of a BPEL process.

BPELUnit can be used as-is for the creation of BPEL unit tests, or extended to add support for new encodings, BPEL engines, and runtime environments. Some remaining research topics have been identified above, and it is believed that this thesis with its generic test framework and a concrete implementation provides a solid basis for both further research and practical use of BPEL unit testing.

Bibliography

ActiveEndpoints. (2006). ActiveBPEL engine. http://www.activebpel.org/ (08/27/2006)

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

Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., et al. (2003). Business Process Execution Language for Web Services, Version 1.1. ftp://www6.software.ibm.com/software/developer/library/ws-bpel11.pdf (08/27/2006)

Apache. (2006a). Apache Ant. http://ant.apache.org/ (08/27/2006)

Apache. (2006b). Log4J. http://logging.apache.org/log4j/docs/ (08/27/2006)

Beck, K. (1999). Simple Smalltalk Testing: With Patterns. http://www.xprogramming.com/testfram.htm (08/27/2006)

Beck, K. (2000). Extreme Programming Explained. Addison-Wesley

Beck, K. (2003). Test-Driven Development by Example. Addison-Wesley Professional

Beck, K., & Gamma, E. (2006). Test infected: Programmers love writing tests. http://junit.sourceforge.net/doc/testinfected/testing.htm (08/27/2006)

Bradby, D. (2006). WSUnit - The Web Services Testing Tool. https://wsunit.dev.java.net/ (08/27/2006)

DAML. (2004). OWL-S 1.1. http://www.daml.org/services/owl-s/1.1/ (08/27/2006)

Dijkstra, E. W. (1970). Notes on structured programming: Technical University Eindhoven

Eclipse. (2006). Eclipse Integrated Development Environment. http://www.eclipse.org/ (08/27/2006)

Ellims, M., Bridges, J., & Ince, D. C. (2004). Unit testing in practice. ISSRE 2004. 15th International Symposium on Software Reliability, 3- 13, 12-15.

Gamma, E., & Beck, K. (2006). JUnit. http://www.junit.org/index.htm (27/08/2006)

Hamill, M. (2004). Unit Test Frameworks. O'Reilly Media

Hong, Z., Patrick, A. V. H., & John, H. R. M. (1997). Software unit test coverage and adequacy. ACM Comput. Surv., 29(4), 366-427.

IBM, Systems, B., Microsoft, AG, S., Software, S., & VeriSign. (2006). Web Services Policy Framework. http://www-128.ibm.com/developerworks/library/specification/ws-polfram/ (08/27/2006)

Juric, M. B. (2006). Business Process Execution Language for Web Services 2nd Edition. Packt Publishing

Leymann, F. (2001). Web Services Flow Language (WSFL 1.0). http://www.ibm.com/software/solutions/webservices/pdf/WSFL.pdf (08/27/2006)

Li, Z., Sun, W., Jiang, Z. B., & Zhang, X. (2005). BPEL4WS Unit Testing: Framework and Implementation. IEEE International Conference on Web Services (ICWS'05), 103-110.

Mackinnon, T., Freeman, S., & Craig, P. (2001). Endo-testing: unit testing with mock objects. In Extreme programming examined (pp. 287-301): Addison-Wesley Longman.

Mayer, P., & Lübke, D. (2006). Towards a BPEL unit testing framework, Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications. Portland, Maine: ACM Press.

McConnel, S. (2004). Code Complete. A Practical Handbook of Software Construction. Microsoft Press

Myers, G. J. (1979). The Art of Software Testing. John Wiley & Sons

Natis, Y. V. (2003). Service-Oriented Architecture Scenario. http://www.gartner.com/DisplayDocument?id=391595 (08/27/2006)

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

OASIS. (2006). Organization for the Advancement of Structured Information Standards. http://www.oasis-open.org/ (08/27/2006)

Oracle. (2006). Oracle BPEL Process Manager. http://www.oracle.com/technology/products/ias/bpel/index.html (08/27/2006)

Predescu, O., & Turner, J. (2006). ANTEater - Ant-based functional testing. http://aft.sourceforge.net/ (08/27/2006)

Stojanovic, Z., & Dahanayake, A. (2005). Service Oriented Software System Engineering: Challenges and Practices. Idea Group Publishing

Thatte, S. (2001). XLANG. http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm (08/27/2006)

W3C. (1999). XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/xpath (08/27/2006)

W3C. (2001). Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl (08/27/2006)

W3C. (2003). SOAP Version 1.2. http://www.w3.org/TR/soap12-part1/ (08/27/2006)

W3C. (2004a). Web Service Architecture. http://www.w3.org/TR/ws-arch/ (08/27/2006)

W3C. (2004b). Web Services Architecture Requirements. http://www.w3.org/TR/wsa-reqs/ (08/27/2006)

W3C. (2006a). Web Services Addressing 1.0 - Core. http://www.w3.org/TR/ws-addr-core/ (08/27/2006)

W3C. (2006b). The World Wide Web Consortium. http://www.w3.org/ (08/27/2006)

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

WS-I. (2006a). Basic Profile Version 1.1. http://www.ws-i.org/Profiles/BasicProfile-1.1.html (08/27/2006)

WS-I. (2006b). WS-I Web Site. http://www.ws-i.org/ (08/27/2006)

List of figures

List of tables

Erklärung

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, September 2006

Philip Mayer