Towards a BPEL unit testing framework

Philip Mayer, Daniel Lübke
University of Hannover, FG Software Engineering
Welfengarten 1
D-30176 Hannover, Germany
+49 511 762 19672
plmayer@acm.org, daniel.luebke@inf.uni-hannover.de

ABSTRACT
The Business Process Execution Language (BPEL) is emerging as the new standard in Web service composition. As more and more workflows are modelled 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 paper, we propose a layer-based approach to creating frameworks for repeatable, white-box BPEL unit testing, which we use for the development of a new testing framework. This framework uses a specialized BPEL-level testing language to describe interactions with a BPEL process to be carried out in a test case. The framework supports automated test execution and offers test management capabilities in a standardized and open way via well-defined interfaces – even to third-party applications.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Testing and Debugging – *Testing tools*

Keywords
BPEL, Composition, Orchestration, Testing, Unit Testing, BPELUnit

1. INTRODUCTION
With the advent of the service-oriented architecture (SOA), formerly proprietary software systems are being opened and made accessible via Web service technology. Web services are software components accessible via the Internet, which can be integrated into more complex, and possibly distributed, applications [2][20].

The Business Process Execution Language (BPEL) [3] is a language for composing Web services. BPEL compositions, described in XML, form an executable program which interacts with other Web services. The composition is recursive, as BPEL compositions are themselves exposed as Web services.

As more and more compositions are modelled using BPEL, ensuring good-quality BPEL code becomes critical. In other areas the need for reliable, automated repeatable testing is already widely recognized, for example in the Extreme Programming community [6] and in the area of Test-Driven Development [7]. Unit testing has already been proven to improve quality in practice [11]. Consequently, there are many unit testing frameworks available for all kinds of programming languages [14].

However, there are still not many efforts for creating unit testing frameworks for BPEL, with the exception of [16]. BPEL editors currently available – like the Oracle BPEL process manager [18], the ActiveBPEL Designer [1] or the preview version of Suns NetBeans 5.5 [19] – focus on manual black box testing, i.e. feeding predefined input data into a BPEL process and comparing the output to a predefined document (or simply presenting it to the user).

However, as BPEL compositions are complex, interacting programs they should be tested like any other software program: Automated, white-box unit testing with the BPEL process as the unit under test and systematic testing of its internal logic can provide valuable feedback and assure quality. In this paper, we explore the realm of BPEL unit testing, proposing a layer-based approach for creating BPEL unit testing frameworks and outlining our implementation.

This paper is structured as follows: First, we introduce the basic ideas for a BPEL unit testing approach in section 2. In the next section, we outline the ingredients of BPEL testing frameworks as well as possible design decisions. Section 4 introduces our own approach to BPEL testing. In section 5, we discuss related work. Finally, we draw our conclusions and give an outlook on further work.

2. TESTING BPEL
BPEL is a language for Web service composition. As such, the most important functionality of a BPEL process is the invocation of other Web services and handling the results from such calls. Internal logic of a BPEL process is often targeted at deciding which Web services to call with which parameters and how to proceed with the results, whether they returned correctly or with a failure (in which case compensation activities take over).

The way BPEL processes communicate with their surroundings – via standard Web service calls – is of great advantage for testing, as opposed to other languages like Java which do not have this kind of unit separation. In fact, defining units when testing Java 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 [7]) to allow separate testing of so-defined units. In BPEL, we get this separation for free: A unit is a BPEL process, and its interfaces are clearly defined through WSDL [20].

TAV-WEB’06, July 17, 2006, Portland, Maine, USA.

Thus, the natural approach to testing BPEL processes is to create an harness around the process under test (PUT), enabling the test 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.

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

3. BPEL TESTING ARCHITECTURE

In this chapter, we present a generic, layer-based approach for creating BPEL testing frameworks, which we later use for the implementation of our own 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 1. The functionality of each layer can be implemented in various ways, which are pointed out in the subsequent sections.

Figure 1: BPEL testing architecture

The first layer is concerned with the test specification – i.e., how the test data and behaviour are formulated. Building on this, the tests must be organized into test cases, test suites, and test runs (test organisation 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).

3.1 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. Such operations specify whether they are one-way or two-way and in which direction the data flows. However, the WSDL syntax lacks a description of the actual protocol involved, i.e. which operation must be invoked after or before which other operation (for a discussion, see [2], 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 all partner processes.

As for the information flow between the BPEL process and its partner processes, we can differentiate 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 provide a way to validate the correctness of incoming data as well as create outgoing data. As pointed out by [16], 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-centred 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/not sending replies. Needless to say, this approach is very simple, but also least expressive.
- **Logic-centred 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-centred 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-centred 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").

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 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.

3.2 Test Organisation

Each test specification contains the test logic and data for one test case. This does not mean all specification artefacts reside in one place, though. A test case could consist of multiple files and/or database entries. The test organisation must provide a way to integrate all these different artefacts into one, representing the complete test case.

However, 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 (so-called test suites). Such organisation has the additional benefit of being able to group tests which require the same setup and shutdown procedures, which then need only be executed once.

Another important aspect of test organisation 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 organisation:

- **Integrated test suite logic:** The first approach is to integrate test organisation with test specification. This is possible only 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 organisation artefacts. These artefacts could include links to the actual test specifications and information about setup and shutdown procedures.

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

3.3 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, instantiated and run upon receipt of a message triggering instance creation. However, for testing a BPEL process there are other possibilities, too.

BPEL process testing means creating a harness (i.e., the test specification) around the PUT, executing the process, and handling input and output data for a concrete PUT instance according to the 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 "mocks" [17] 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 standardised. 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 standardised as well. Furthermore, the BPEL specification allows dynamic discovery of partner Web services. Although frequent use of such features is doubted [2], a framework relying on real-life test execution will have no way to counter such URI replacements.

There are certain correlations between the two approaches discussed in section 3.1 and the two execution types. Indeed, the choice of specification has a strong influence on the way the test should be executed. 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.

3.4 Test Results

Execution of the tests yields results and statistics, which are to be presented to the user at a later point in time. Many metrics have been defined for testing [22], 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 JUnit [13]: 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 flavours. 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; which 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 standardised 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 are again 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 most PUT activities (i.e. all activities which deal with outside Web services, which are in fact most of the activities). 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.

With this explanation of the test result layer, we have finished our description of the four-layer BPEL testing framework architecture. In the next section, we present our own instance of this generic framework.

4. BPELUnit – A BPEL TESTING FRAMEWORK

Section 3 provided an overview of the possible architectures of BPEL testing frameworks and presented some of the choices to be made when instantiating such a framework. In this section we present our own framework design. We describe the choices made for each of the framework layers and our implementation approach.

4.1 Framework Design

In this section, we describe our implementation choices for each of the four layers of the testing architecture.

4.1.1 Test Specification

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

Our framework is aimed at "test-infected" developers [12], 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 raises two questions:

- How to specify the **data**, i.e. at what level (the lowest possible level being SOAP, and the highest possible level depending on the implementation of the corresponding Web service).
- How to specify the **interaction details**, i.e. what protocol to expect at which partner.

BPEL Web services are based on WSDL descriptions which use XML Schema for type definitions (see [3], Chapter 1). When creating BPEL processes, developers thus deal with XML variables and message formats which can be validated against a given XML Schema. The so-defined XML is therefore the "natural data language" for BPEL, and we believe it is also the best language for specifying data to be sent to the PUT.

One could use the same format to check incoming messages – i.e., compare the XML node-by-node. However, messages from the PUT may contain random data like dates or auto-incremented numbers, which may not even be relevant for the test. Instead of specifying which data is not relevant in a complete message, we adopt the opposite approach: Specifying which data is relevant by means of XPath expressions [10]. An incoming message can then be checked against one or more Boolean XPath expressions, thus making sure it contains all the relevant details.

With the data format specified, we can move on to the interaction details. As the BPEL process is a Web service cooperating with other Web services, the following interactions can take place:

- **One-Way** (Receive Only and Send Only). Although probably rarely used, the combination of these two interactions can be used for fire-and-forget calls.
- **Two-Way Synchronous** (Send/Receive and Receive/Send). These are the most obvious interactions. The first one will be mainly used in the client of the PUT, whereas the second will be mainly used by partners.
- **Two-Way Asynchronous** (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.

Testing the PUT means verifying the correctness of the implemented business protocol. To do this, one needs to simulate the business protocol of the client and the partners to provoke and test the reactions of the BPEL process. We will thus allow testers to specify sequences of the interactions mentioned above for the client as well as every partner of the PUT. By chaining these atomic interactions together, it is possible to easily shape different interaction protocols with the PUT on a case-by-case basis.

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 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.

By using XML data and sequences of atomic interactions with the PUT for the test data specification, our 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 for the WSDL messages, 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 testing.

Due to the simplicity of the language, the actual execution is not very difficult for the framework. Tool support for creating the test specification is also possible. For every interaction, a wizard may be created which allows the user to fill in all the relevant details; it is also possible to create an XML-Schema-based UI to generate and edit the actual data to be sent.

4.1.2 Test Organisation

As defined in our 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).

Our test organization combines multiple test cases with a setup part containing links to the external artefacts, thereby creating a test suite. The test organization is thus separate from the test specification, although they may reside in the same file.

BPEL testing is different from other xUnit approaches in that 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 setup part of our test suite document contains all necessary information to execute these phases.

By using an XML format for the test suite, it is also easily possible to augment the test cases and test suites with additional metadata, like for example the requirement tracing data mentioned in chapter 3.

This approach leaves us with a central access point to the test – the test suite specification, which is used as a starting point for test execution.

4.1.3 Test Execution

Both available choices for testing the PUT – simulated and real-life testing – are dependent on some sort of API or deployment descriptor in the engine, which means vendor-lock-in, as there are no standards in this area.

To keep our framework free from such dependencies – and therefore maximally general – we tried to use an approach which would allow us to decouple from concrete engines by writing adapters. The easiest way to do this is to create a wrapper around the deployment process (possibly including the generation of deployment descriptors) for each particular engine.

Therefore, we have adopted the real-life deployment approach: deploying the PUT before the test, running the test, and undeploying the PUT afterwards.

4.1.4 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 in chapter 3, one way of gathering 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 of the engines to query the engine about the state of a process instance after its completion. We hope to be able to leverage such APIs across engine vendors, but this is still subject to further research.

4.2 Implementing the framework

Implementing the BPEL testing framework as laid out in section 4.1 first requires us to define an adequate format for the test suite document, i.e. a way for developers to define test cases and deployment information. Our approach is described in section 4.2.1.

Afterwards, we outline the software design of the core framework which handles test execution and gathering of results. The design takes arbitrary WSDL styles and encodings, multiple BPEL engine vendors and UIs into consideration and is described in section 4.2.2.

4.2.1 Writing tests

As already hinted at in section 4.1, the easiest way of integrating test data, interactions, and deployment information for a test suite is to use an XML-based format. Our test suite document consists of two parts:

- The first part is the deployment section, in which the PUT and all of the partners are specified
- The second part is the test case section, which contains an arbitrary number of test cases

In the following two sections, we will discuss each part.

4.2.1.1 The Deployment Section

The deployment section specifies the simulated partners along with their WSDL files, and contains information on how to deploy and undeploy the PUT. As our framework is extensible to be used with different engines, the deployment information is laid down in a vendor-specific way and passed on to the deployer registered for the given PUT type.

We will provide deployers for the Oracle BPEL server [18] and the open-source BPEL engine ActiveBPEL [1]. Our framework provides extension points which can be used to provide support for other engines. Adding support for an engine requires programmatic (un-)deployment support and specification of the deployment tags for the test suite document.

Running a test suite means deploying the PUT, setting up the framework for sending and receiving calls on behalf of the client and partners, running the test cases, and undeploying the PUT after the test. The deployment section is thus used both for set up and shut down of the suite.

```
<deployment>

  <put type="oracle" name="TravelDoc">

    <deploymentOptions>

      <bpd:oracleDeployment
          xmlns:bpd="http://www...oracle"

        processName="TravelDoc"
        compiledBPELJarFile=
          "bpel_TravelDoc_1.0.jar"

        domain="default"
        password="bpel"  />

    </deploymentOptions>

    <wsdl>TravelDoc.wsdl</wsdl>

  </put>

  <partner name="Employee"
    wsdl="EmployeeDatabase.wsdl"/>

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

</deployment>
```

Figure 2: Deployment Section

Figure 2 shows the deployment section for the Business Travels Web Service example from [15]. The PUT specification contains all the necessary details for deployment in an Oracle-specific format, which will be passed on to the deployer registered to the type `oracle`. It also contains a link to the WSDL file of the PUT.

Additionally, two partner Web services are specified, each with a link to the implemented WSDL. The names of the partners will later be used in the test case section.

4.2.1.2 The Test Case Section

The second part of the test suite document contains the test cases. Each test case contains one thread for each partner, called a partner track, which contains a sequence of interactions, called activities, which describe expected actions from the PUT or actions the partner must take.

Figure 3 shows an example test case section. Two partner tracks are specified – one for the client and one for a partner called `Airline`. Each partner track contains a sequence of activities.

As can be seen in the example we also define a property-based extension mechanism for each test case and test suite, 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 (see section 4.2.2).

```
<testCases>

  <testCase name="Travel Test">

    <property name="useCase">245</property>

    <clientTrack>
       ...
    </clientTrack>

    <partnerTrack name="Airline">
       ...
    </partnerTrack>

  </testCase>

</testCases
```

Figure 3: Test Organisation Example

As an example of an activity, Figure 4 shows a synchronous send/receive inside a test specification for the Business Travels Web Service example from [15]. This is an activity for the client of the PUT, instructing it to send the specified data to the PUT, wait for a synchronous answer, and verify the answer according to the given condition.

```
<clientTrack>

 <sendReceive
    service="travel:TravelDoc"
    port="TravelDocPort"
    operation="process">

  <send>

    <data>
     <travel:TravelDocProcessRequest>
      <travel:employeeData>
        <emp:FirstName>Philip</emp:FirstName>
        <emp:LastName>Mayer</emp:LastName>
        <emp:Department>SE</emp:Department>
       </travel:employeeData>

       <travel:flightData>
          ...
       </travel:flightData>

     </travel:TravelDocProcessRequest>
    </data>

  </send>

  <receive>

    <condition>
       travel:TravelDocProcessResponse/
              aln:Approved[1]='true'
    </condition>

  </receive>

 </sendReceive>

</clientTrack>
```

Figure 4: A synchronous send/receive

The framework assists the tester in detecting the three kinds of incoming data errors listed in section 3.1. Incorrect message data is easy to detect: the XPath conditions in the receive block 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 receive block for a message, a fault is generated.

Figure 5 shows a sequence diagram of a typical interaction between the framework, its client and partner tracks, and the PUT. The PUT is a simple BPEL process which delegates a credit request to two partners.

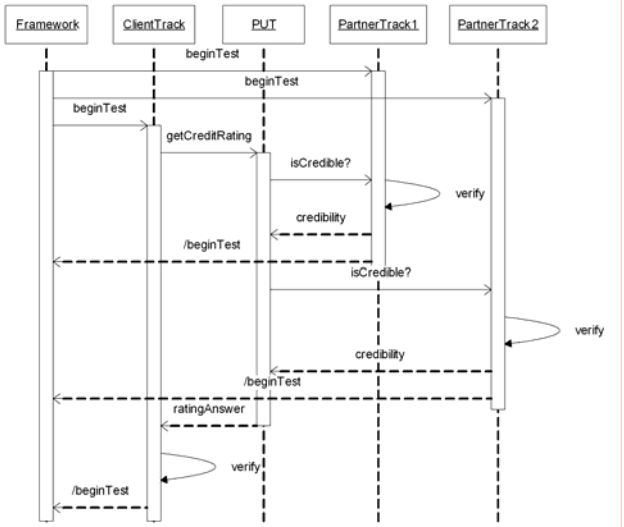

Figure 5: A Testing Sequence

When thinking about the specification of data and interactions for use in a real-life setup, one has to take two additional challenges into consideration: the different possible SOAP styles and encodings, and how to address partners in asynchronous calls.

4.2.1.2.1 Handling SOAP styles and encodings

The XML data specified in the test suite document is plain XML-Schema-based XML as specified in the type definitions for a WSDL message. However, this data may be very different from the SOAP data on the wire. The wire format depends entirely on the concrete style and encoding of the given WSDL binding. For example, in the case of a document/literal encoding the SOAP envelope is merely a very thin wrapper around the literal XML data; in the case of RPC/literal, there are even more wrappers, and in other encodings, the whole message may change.

In our opinion, a tester should not need to concern himself with the actual wire format of the messages. This is a task best left automated, and our framework will automatically create the wire format required for each partner Web service and the PUT itself according to the selected bindings.

To allow this kind of operation, the specification of each activity includes links to the service, port, and operation (see Figure 4). The framework uses this information – among other things – to extract the SOAP message style and encoding from the WSDL, which is required for correctly encoding or decoding a message sent to or received from the PUT. As pointed out in chapter 4.1, the WSDL may specify arbitrary styles and encodings. Our framework contains encoders for the two styles allowed by the WS-I Basic Profile [5] (document/literal and rpc/literal), but contains an extension point to plug-in arbitrary encoders for other formats.

4.2.1.2.2 Handling asynchronous addressing

When using synchronous send/receive or receive/send operations and the HTTP transport, the respective answer can simply be returned in the same connection, and no special addressing is needed. However, when using asynchronous messaging, a separate HTTP request must be used for the call-back, and addressing information for this request is required.

Where and how to specify call-back information is basically up to concrete Web service implementations. Therefore, the framework uses an extensible mechanism to allow arbitrary addressing-related manipulation of SOAP headers before a message is sent and right after it is received. We provide an implementation of the WS-Addressing specification [8]; other processors may be registered with the framework by means of an extension point.

The concrete addressing implementation to use is specified by the tester using a special tag inside of asynchronous activities.

4.2.2 Framework Core Layout

Consistent with many BPEL engines and the goal of platform-independence, our 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, and possibly, the engine.

Figure 6 shows the design of the framework. The core provides an API and three extension points for external software.

The API is intended to be used by clients of the framework to present a UI to the user and as such allow execution of tests. Examples of such clients are displayed in the figure – a command line client, an ant integration library, and a plug-in for Eclipse.

At the extension points, adapters can plug in to offer deployment for a particular BPEL engine, encoding support for a particular WSDL/SOAP style and encoding, and header processors for a particular addressing mechanism.

- A BPEL deployer is registered with a type (or name). If this type is used within a test suite specification, the corresponding deployer is instantiated. At the beginning of the test run, the deployer is instructed to deploy the process with the deployment settings from the test suite specification. After the test cases have been run, the deployer is instructed to remove the process from the engine.
- An encoder is responsible for encoding to and decoding from a particular SOAP message format. An encoder is registered with a certain style and binding. If that combination is encountered in a WSDL file by the framework, the encoder is instantiated. When a message is about to be sent, the encoder is responsible for converting the literal data into a complete SOAP message. When a message is received, the encoder is responsible for retrieving the literal data from inside the SOAP message.
- A header processor is registered with a name, which allows it to be referenced from inside an activity, where the tester

specifies the processor required for a certain operation. Header processors are free to change the SOAP header in any way they see fit, although the main objective is to allow call-back addressing.

On the right-hand side in Figure 6, the user-written test specification is displayed, which in turn contains links to the user-written BPEL process. Note that the BPEL code just passes through the core, processing it is not required.

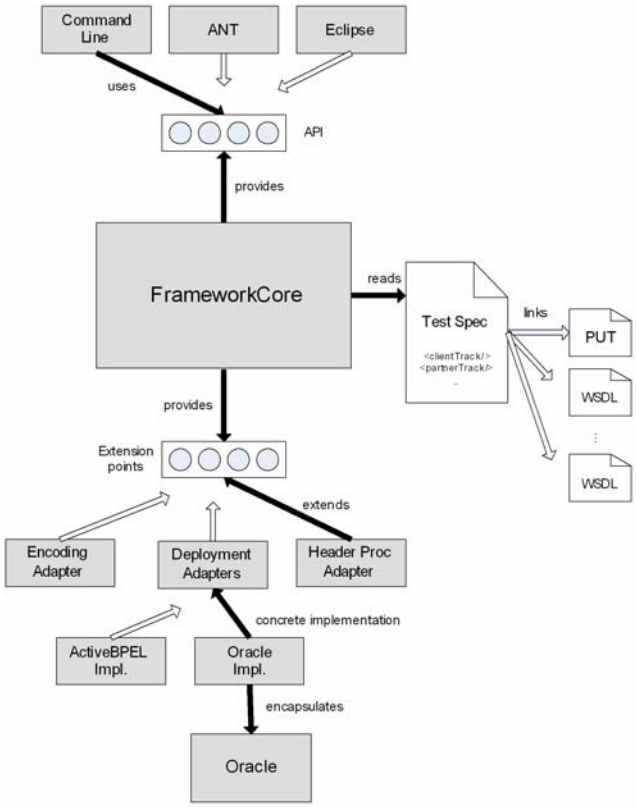

Figure 6: Framework Core Layout

The framework is invoked with a test suite document via the UI APIs. It passes the information on to the selected deployment implementation, which deploy the linked process. The partner tracks are then instructed to start the test. The framework waits for the tracks to complete (throwing a fault, or returning normally). Afterwards, the selected deployer is instructed to undeploy the BPEL process, and the next test suite may be run.

As an example of a UI, Figure 7 shows the Eclipse test runner which deliberately looks like the JUnit test runner already supplied with Eclipse. A test run may be started inside Eclipse by creating a BPELUnit launch configuration for a test suite specification file. During the launch, the BPELUnit framework core is instantiated and instructed to run the test suite. During and after the run, the progress and results of the run are presented to the user.

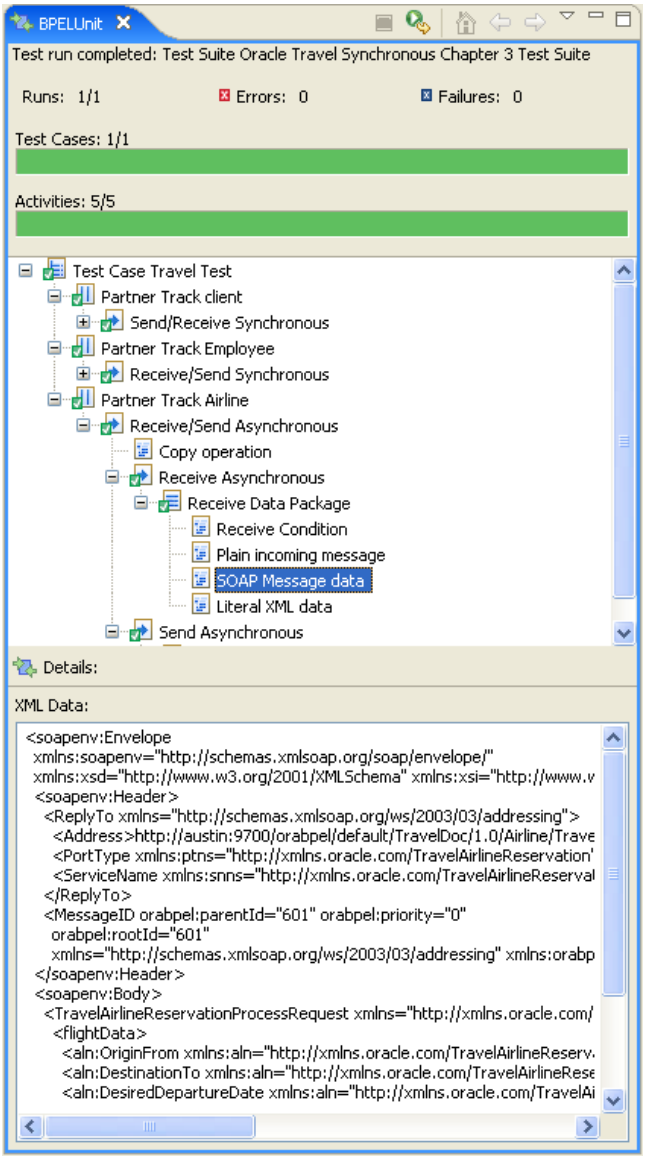

Figure 7: BPELUnit Eclipse View

The BPELUnit view differs from the JUnit view in the following ways:

- The tree view does not only show the test suite and test cases, but offers a deep view into what is going on inside the partner tracks and activities. This is of particular importance as there are nearly unlimited sources of error when dealing with remote calls. The test runner shows a complete history of every message sent or received, from literal data to complete SOAP message.
- Instead of a Java stack trace, the detail pane shows more information about a selected test artefact. In the figure, a complete SOAP message is shown; selecting activities would yield more information about how the activity was executed.
- There are two progress bars; one for the test cases and one for the activities of the current test case, which allows better progress tracking.

4.3 Tool Support

Although the test specification is very simple, writing the XML document can be tedious work for the developer. Therefore, we propose tool support for aiding the developer in creating both the test interaction details as well as the actual test data.

The first question to be answered when contemplating tool support is the intended target group, i.e. the users of the tools. On the one hand, BPEL is a programming language which allows writing rather complex programs. On the other hand, BPEL compositions are fairly high-level artefacts; close to business goals and requirements of the resulting system. Two different user groups of BPEL testing tools can therefore be anticipated:

- **BPEL developers**: Naturally, there must be someone who wrote the original PUT BPEL code, and as suggested by some [6][7], testing and development should be interleaved. Therefore, one option is to create tool support for the developer himself.
- **Specialized testers**: Another way of approaching testing is to entrust testing to a specialized department consisting of professional testers. These testers are closer to requirements than they are to code; therefore the tool support must concentrate on requirements, too.

In the case of BPEL developers as the target group, the tool support probably should consist of creating skeletons for the test specification document, which are then to be filled by the developer. In the case of specialized testers, the tool support could provide wizard-based test generation tools based on more simple representations of the test data. This, however, will not be possible for all types of PUTs and interactions. It will be an interesting challenge, though, to see how far this approach scales.

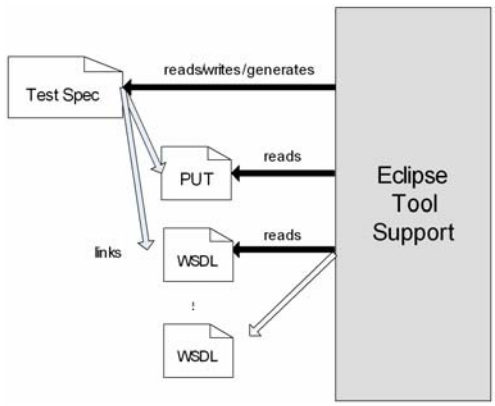

Figure 8: Eclipse Tool Support

The tool support for BPEL testing is currently under research and has not yet been implemented. Figure 8 shows a basic architectural diagram of the tool support, which will be implemented as Eclipse plug-ins.

Note that the actual framework and the tool support are independent of one another and only connected through the test specification and the linked PUT and WSDL. Therefore, other forms of tool support are easily implemented on top of the framework.

5. RELATED WORK

As the BPEL language is relatively new, there are still not many efforts for creating unit testing frameworks specifically targeted at BPEL. The area of BPEL testing is currently restricted to either theoretical approaches [16] or practical embedded approaches (as implemented in the ActiveBPEL designer, or NetBeans 5.5).

- The approach in [16] 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. Our approach differs from [16] 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. 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 respective BPEL engines run in a "simulation mode", which allows the testing framework to directly inject or extract data instead of making actual SOAP test calls. These approaches focus on manual testing, are limited to a particular engine and also do not test the complexities involved in the SOAP encoding process as well as the message transport.

As BPEL processes are Web services, existing web-service testing tools can also be used for BPEL testing. However, most of these testing tools regard Web services as black boxes, only to be instrumented by a client and without simulating possible partners. Simulating a partner is in many ways opposite to client-side testing of Web service, as the testing tool must simulate a Web service instead of interacting with it, for example by extracting and using addressing information for not only receiving asynchronous call-backs, but actively creating and sending them.

As an example of such tools, we discuss WS-Unit [21] and ANTEater [4], both of which are open source. WS-Unit is a "Web service consumer tester", i.e. it can be used to simulate a BPEL partner. ANTEater, on the other hand, is a functional testing tool, intended for simulating a client (including asynchronous call-backs). Both tools also allow simple copying and verification rules like the ones we use in our framework. However, our framework differs in two important points from these tools:

- We are using 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.
- Instead of focussing on single interaction sequences, our framework allows the specification of parallel threads of activities required for simulating several partners of a PUT at once, and also provides the ability to extract call-back information to create server-side call-backs.

6. CONCLUSION

In the first part of this paper, we have presented a generic layer-based approach to creating testing frameworks for repeatable white-box BPEL unit testing. Each layer has been described systematically and several implementation techniques have been proposed. We believe that the given architecture can be used for classification of existing frameworks and as an aid for future implementations of BPEL testing frameworks.

In the second part of this paper, we have presented our specific implementation of this framework, which uses literal XML data and a custom interaction mini-language for the test specification as well as a Java-based test runner which can be extended to support multiple BPEL engines, SOAP styles and encodings, addressing modes and UIs for test execution and result presentation. We believe that our approach greatly facilitates BPEL testing, as it is easy to use, operates on the same data level as BPEL, and provides extensive reporting capabilities inside the Eclipse UI.

Due to a flexible architecture, our BPEL testing framework is usable both on the client side by the developer himself, for example inside an IDE such as Eclipse, as well as on the server side, for example by using Ant for running all tests as part of a nightly build.

As outlined in the previous chapters, our BPEL testing framework is still a work in progress. We will continue working on the question of gathering metrics from the process under test to create coverage figures, and on tool support for aiding both programmers and testers in creating BPEL test cases. Especially the integration into development environments including tool support is an important aspect; our implementation will feature integration into the Eclipse IDE. The presented concepts for a BPEL unit testing framework will serve as a solid foundation for our future research.

7. REFERENCES

[1] ActiveBPEL Engine. http://www.activebpel.org/

[2] Alonso, G., Casati, F., Kuno, H., Machiraju, V. Web services. Springer-Verlag Berlin Heidelberg, 2004.

[3] Andrews, T., Curbera, F., et al. Business Process Execution Language for Web services 1.1. July 2002. http://www-128.ibm.com/developerworks/library/specification/ws-bpel/.

[4] ANTEater. Ant-based functional testing. http://aft.sourceforge.net/

[5] Ballinger, K., Ehnebuske, D., et al. WS-I Basic Profile Version 1.0. http://www.ws-i.org/Profiles/BasicProfile-1.0.html

[6] Beck, K. Extreme Programming Explained. Addison-Wesley, 2000.

[7] Beck, K. Test-Driven Development by Example. Addison-Wesley, 2003.

[8] Box, D., Christensen, E., et al. Web Services Addressing (WS-Addressing). http://www.w3.org/Submission/ws-addressing/

[9] Christensen, E., Curbera, F., Meredith, G., Weerawarana, S. Web Service Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl

[10] Clark, J., DeRose, S. XML Path Language Version 1.0. http://www.w3.org/TR/xpath

[11] Ellims, M.; Bridges, J.; Ince, D.C., "Unit testing in practice," Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on Software Reliability Engineering, pp. 3- 13, 2-5 Nov. 2004

[12] Gamma, E., Beck, K. JUnit Test Infected: Programmers Love Writing Tests. http://junit.sourceforge.net/doc/testinfected/testing.htm

[13] Gamma, E., Beck, K. JUnit. http://www.junit.org/

[14] Hamill, P. Unit Test Frameworks. O'Reilly, 2004

[15] Juric, M. B. Business Process Execution Language for Web Services Second Edition. Packt Publishing, 2006.

[16] Li, Z., Sun, W., Jiang, Z. B., and Zhang, X. 2005. BPEL4WS Unit Testing: Framework and Implementation. In Proceedings of the IEEE international Conference on Web services (Icws'05) - Volume 00 (July 11 - 15, 2005). ICWS. IEEE Computer Society, Washington, DC, 103-110.

[17] Mackinnon, T., Freeman, S., and Craig, P. 2001. Endo-testing: unit testing with mock objects. In Extreme Programming Examined, G. Succi and M. Marchesi, Eds. The XP Series. Addison-Wesley Longman Publishing Co., Boston, MA, 287-301.

[18] Oracle BPEL Process Manager. http://www.oracle.com/technology/products/ias/bpel/index.html

[19] Sun NetBeans Enterprise Pack. http://www.netbeans.org/products/enterprise/index.html

[20] Weerawarana, S., Curbera, F., Leymann, F., Storey, T., Ferguson, D. Web services Platform Architecture. Prentice Hall PTR, 2005.

[21] WS-Unit. The Web Service Testing Tool. https://wsunit.dev.java.net/

[22] Zhu, H., Hall, P. A., and May, J. H. 1997. Software unit test coverage and adequacy. *ACM Comput. Surv.* 29, 4 (Dec. 1997), 366-4