6

Unit Testing BPEL Compositions

Daniel Lübke

Leibniz Universität Hannover, FG Software Engineering, Germany
daniel.luebke@inf.uni-hannover.de

Abstract. Service-Oriented Architecture is a new emerging architectural style for developing distributed business applications. Those applications are often realized using Web services. These services are grouped into BPEL compositions.

However, these applications need to be tested. For achieving better software quality, testing has to be done along the whole development process. Within this chapter a unit testing framework for BPEL named BPELUnit is presented. BPELUnit allows unit and integration tests of BPEL compositions. The tester is supported as much as possible: The used Web services can be replaced during test execution. This allows to really isolate the BPEL composition as a unit and guarantees repeatable tests.

6.1 Introduction

Service-Oriented Architecture (SOA) has become an accepted architectural style for building business applications. The application's logic is decomposed into fine-grained services which are composed into executable business processes. Services in SOA are loosely coupled software components, which most often offer functionality in a platform-independent and network-accessible way. Therefore, SOA is a *functional* decomposition of a system.

SOA aims to better align business processes and their supporting IT systems. Thus, changes within the processes should easily be incorporated into the IT infrastructure. Using fine-grained services, compositions can be updated easily by rearranging said services – hopefully without the need to change the services themselves. In this scenario, services can be offered by internal IT systems or can be bought from external service providers or partner organizations. This way, it is possible to integrate IT systems from different enterprises, e.g. in order to optimize supply chains or building virtual organizations.

While SOA as an architectural style is not dependent on any technology, the dominant implementation strategy is to use Web service standards. Supported by all major software companies, Web services and their related

technologies, like the Business Process Execution Language (BPEL), have relatively good development support despite being a new technique.

BPEL is used for composing Web services into complex business processes. It supports rather complex programming constructs. These are the same as in normal programming languages, e.g. conditions, loops and fault handling. Therefore, BPEL compositions are software artefacts as well, possibly containing complex logic which is error-prone.

Being aware that BPEL compositions are subject to the same problems as normal software, it is necessary to test them in order to find as many defects as possible. This is especially necessary since BPEL compositions are normally deployed at high-risk positions within a company. However, testing BPEL is problematic due to its nature: BPEL compositions have many external dependencies, namely the Web services it accesses.

While non-functional testing has attracted much attention within the research community, functional testing of service compositions can be problematic as shown in this chapter. To address this issue, this chapter presents a unit testing framework for BPEL processes called BPELUnit. The framework was developed to ease the burden of testers and programmers in BPEL-related projects by allowing Web services to be mocked at run-time.

The next section of this chapter will shortly categorize services before some problems special to testing service compositions are presented in Sect. 6.3. Afterwards, different test types, which developers and testers will face in SOA projects, are described in Sect. 6.4. Section 6.5 describes a generic layered architecture for test tools, especially unit testing frameworks. This architecture has been used to develop BPELUnit – a unit testing framework for BPEL compositions – which is presented afterwards. The last section illustrates the difficulties in testing compositions presented during this chapter by giving a short example.

6.2 Service Classification

Web services are categorized within this chapter by two dimensions: The organizational dimension and the composition dimension as shown in Fig. 6.1.

A service can be developed and deployed by the organization itself or can be offered by an external party. Examples for internally developed services are wrappers around legacy systems and custom created services. Source code for

	Atomic Service	**Composed Service**
Internal	Legacy Wrapper Custom Services	BPEL Orchestrations
External	Network-accessible Service hosted outside the organization	

Fig. 6.1. Categorization for services

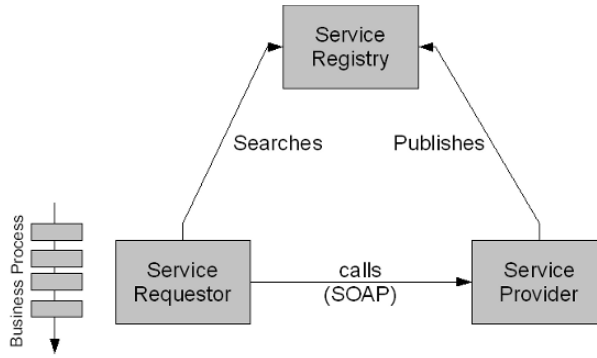

Fig. 6.2. Distributed roles in SOA make testing more difficult

such services is available and generally they can be replicated in a testing environment. In contrast, many services are available in the Internet, which can be used by applications without knowing how the services work. Consequently, such services cannot be replicated within a testing environment.

The organizational dimension can be seen in the SOA triangle as illustrated in Fig. 6.2: SOA systems span multiple roles, most important the service provider and the service requester.

The other dimension is (visible) service composition: Services can either be atomic, i.e. provided as such, or put together by composing other services. The former ones are either implemented in traditional programming languages like Java or are provided as is without access to the services' logic. The latter are processes composed in-house using composition languages like the Business Process Execution Language (BPEL).

It is notable that there is no composed, external service from an organization's point of view: Since the service is external, it cannot be accessed, modified nor installed locally. It is a black box and therefore it is irrelevant for the tester in which way the service is implemented. Such external services are an obstacle in testing: Normally, they cannot be used intensively for testing, because they need to be paid for or no test account is available. However, integration and system tests – as described in Sects. 6.4.3 and 6.4.4 – are only possible if the original services can be used.

In contrast, when unit testing BPEL compositions, all atomic services can be mocked, i.e. replaced by dummy services. This allows the composed services to be tested in isolation and without the need for external dependencies.

6.3 Challenges in BPEL Testing

Testing software is a time-consuming and often neglected task. Testing BPEL compositions is even harder due to the following reasons:

- Test performance: BPEL compositions depend on calling Web services. SOAP calls are extremely costly due to intense XML parsing and often

associated XSL transformations, as well as network overhead. Example measures are, e.g., given by [9].
- Error conditions: Due to the distributed nature, many errors must be expected and handled by the system. For example, networking errors and temporarily not reachable services need careful error handling which needs to be tested too.
- Dependencies: Web services represent external dependencies for the BPEL composition. The composition relies on the correct behaviour of the services in order to fulfil its tasks.
- Deployment: BPEL compositions need to be transferred on a BPEL server and be made ready for execution. This process, called deployment, requires time-consuming preparation before testing can start.
- Complexity: BPEL compositions normally contain many elements, like assigns and Web service calls. Especially, the use of complex XPath queries and XSL transformations lead to many cases which need to be tested.
- Organizational borders: As already outlined, SOA systems can span multiple organizations, which do not share their service implementations. This hinders setting up a test environment as outlined in Sect. 6.2.

Because BPEL is a relatively new technique, testers and developers do not have much experience in which areas defects are likely to occur. This reduces test efficiency until testers are able to get a "feeling" in which areas they are likely able to find defects.

All these factors require intensive research into testing habits and test support for service compositions in general and for BPEL in particular.

6.4 Test Types for Compositions

6.4.1 Test Levels

Software testing is a widely used quality-improvement activity, supported by both academic research and commercial experience. In his timeless classic, *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" ([15]).

While newer definitions are available, this simple but precise definition hints at the attitude a software tester should have: He or she should not try to prove that a software works correctly (i.e. has no defects), but rather to find cases in which it does not work (i.e. has defects). The former is impossible anyway – as pointed out by [3]: Testing can only show the presence of errors, but not their absence.

There are many different forms of software testing, each on a different level addressing different error types (see also [14]):

- 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. This type of tests try to detect errors inherent to one unit, like wrong logic etc.

- Integration testing: An integration test is used to test combined, or integrated, components of a software system. Such tests try to spot errors which are introduced by the combination of different components, e.g. differently interpreted interfaces etc.
- System testing: System testing is the process of testing the complete, final system, which includes interactions with other systems and all components. Such tests are typically done at the end of an iteration. Using this kind of tests, projects try to find any fatal errors before delivering the software.

All types of tests can be automated. Automated test are often used in regression testing, which can therefore be repeated easily and cheaply. Regression testing intends to find bugs in updated software which previously has already passed the tests.

An extreme form of automated testing is Test-First. Test-First has established itself as a new way of creating test cases before code is written. Especially successful in Agile Methods, like Extreme Programming ([2]), it has shown its ability to increase the quality of the tests, e.g. in ([5, 4]).

In the following course of this section, the different kinds of tests are described more precisely. This includes special problems service compositions raise in these corresponding contexts.

6.4.2 Unit Tests

As pointed out above, unit testing is the process of testing a single component, named a unit, of a program in isolation. It has been wholly embraced by the Extreme Programming community ([2]) and in the area of Test-Driven Development.

In the context of service compositions and BPEL, unit testing implies that all Web services, as they represent external dependencies, need to be replaced by mocks. However, those mocks need to be implemented and deployed which can be a rather time-consuming task. Furthermore, in order to develop Web services (e.g. in Java and .Net), programming skills are needed which are not necessarily available in a testing team.

The problem of mocking external services is one of the most important drivers for tool automation for unit tests. Even in other languages like Java there are mocking facilities available ([7]). However, in the world of Web services, tool support is more critical due to the mentioned reasons. Because of this, mocking support has been incorporated into the unit testing framework itself rather than being an independent component.

While other languages have support for unit tests by special frameworks, like JUnit for Java ([8]), BPEL lacks such support. Therefore, one of our research goals was to develop unit test support for BPEL, which is described in Sect. 6.6.

6.4.3 Integration Tests

Integration testing concerns two or more components. Ideally, these components are tested using unit tests, so that the functionality can be anticipated to work (mostly) correctly.

Integration testing tries to verify the interaction between several components. The tests try to trigger all communication and calls between the components.

Integration testing in service composition is mostly an organizational challenge: For testing compositions all services need to be available. Normally, a testing environment is set-up, in which services, databases etc. are replicated and can be used and their contents changed. However, when using an externally provided service, like a payment service, it is impossible to install the same service on-site. Instead, there are essentially two possibilities during integration testing, whenever a test environment is not available:

1. Mocking of external, atomic services: External services are mocked as they are in unit tests. This has the advantage that communication between all self-developed components can be done at all times during the development process. However, the interaction between the self-developed parts and external services cannot be tested this way. This option is only applicable for testing two compositions depending on each other, so that their combination can be tested without the need of the respective dependent services.
2. Test accounts: The service provider may offer test accounts, e.g. a dummy environment for a CRM system may be offered. This environment can be used for testing purposes by the testers.

It is notable, however, that this problem only arises when services store data. Whenever a service only calculates data or returns values, like search engines, the original service can normally be used without any problems.

In case of non-free services, for which the consumer has to pay, integration tests should be optimized for using as few calls to services as possible. Mocking service calls in non-essential situations may be an option too.

Integration testing is especially important in Web service–based projects, since WSDL descriptions of Web services only define the syntax of the services but neglect semantic aspects like meaning of parameters, fault behaviour etc.

6.4.4 System Tests

At the end of an iteration or the project, the system is tested in its entirety. Therefore, the whole system is normally installed in a test environment, replicating the real environment in which the application will run later on. This will include the composition engine, e.g. the BPEL server, and the developed services. As with integration tests, the problem during system test is the replication of external services. However, during system test it is unacceptable to

mock certain services. If services cannot be replicated internally, a test account must be created by the service provider or the test team needs to utilize the final configuration.

Dynamic discovery of services poses a significant problem for system testing: If a service is not bound statically to the composition, the composition engine tries to discover, e.g. by using UDDI, a matching service. The selected service can change over time. However, the new service has not been part of the system test, possibly rendering its results worthless.

6.4.5 Test-First

Test-First is not directly a class of testing as unit integration and system tests are. Instead, it describes a philosophy of development: Tests are written in small chunks before actual code is written. Test-First is an inherent practice in Extreme Programming and Test-Driven Development. For service compositions, this means that a part of the process is specified before development as a test case. Afterwards, the composition is written, e.g. one service is called and the correspondent variable assignments are made. Finally, the tests are complemented with tests for error handling and new functionality and the composition is updated to fulfil the tests. Hereby, all external dependencies are excluded and mocked as well. These steps continue until all requirements and error handling routines have been developed.

Especially with composition projects, in which not all services are initially available to the development organization, Test-First is a good option to start development: All external references to unavailable services can be replaced by dummy services called mocks. The missing services can be integrated later and are immediately ready for integration testing, which will try to detect misunderstood interfaces.

For Test-First, test automation is very important. Since tests are run after every little implementation step, manual testing is too cumbersome and time-consuming. Therefore, unit testing frameworks are a necessity in test-driven projects.

6.4.6 Remarks

Testing service compositions is comparable to testing "normal" software systems. The same types of tests can be integrated into the development process. However, their relevance changes: Unit testing compositions is easier than in traditional programming languages, since all parts of the system are already loosely coupled and can be replaced by mocks. Mocks play a special role in testing compositions since they can replace external services in all types of tests whenever the use of original services is impossible or at least too costly. This hints at the major problem: Replicating the system in a test environment is often impossible whenever services are hosted externally. The testing team

should try to mitigate these problems early and try to replace the services or have special accounts not interfering with the production environment.

Because of the special role of using mocks and trying to isolate parts of the system over which the development organization has control, tool support is necessary. Only by using adequate tools, compositions can be easily and efficiently isolated and mocks created accordingly.

Furthermore, since service compositions, especially BPEL, are normally geared towards machine-to-machine communication, automation is a desirable goal: Repeated tests using manually entered XML-data are enormously expensive and time-consuming. Additionally, all XML artefacts need to be managed. Accordingly, tools should be able to handle and store the large and, for humans, often unreadable XML data.

One available tool for generating stubs is WSUnit ([6]). However, it lacks integration into the other test tools: WSUnit needs to be deployed in a J2EE web container before it is used and cannot detect whether the values passed to it are correct and consequently abort the test run. The deployment has to be done before the tests are run and therefore needs to be integrated into an automated test run.

6.5 Testing Architectures

In this section, a generic, layer-based approach for creating BPEL testing frameworks is presented, which is later used for the design of the BPELUnit framework [13]. As a side effect, this layer-based model can be used for classifying existing frameworks or implementations of other frameworks.

Testing tools can be geared towards different roles in development projects which consequently have different requirements, usage models and technical backgrounds. For example, the "test infected developer" doing Test-First on his BPEL composition is a technical savvy person, understanding XML data and has knowledge in SOAP. He or she wants to write tests efficiently and run them every five minutes. However, a pure tester does not want to deal with technical details. He or she most likely does not need mocking support but wants to organize a large number of test-cases. Therefore, design decisions must differ for the intended target group, but the layered architecture can be the same for all.

The proposed architecture consists of several layers which build upon one another, as outlined in Fig. 6.3. The functionality of each layer can be implemented in various ways, which are shortly pointed out in the subsequent sections.

The first (bottom) 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 suites, which is the responsibility of the test organization layer. A test – and therefore also the process under test – must be executed. This task is performed by the test execution layer. During the test

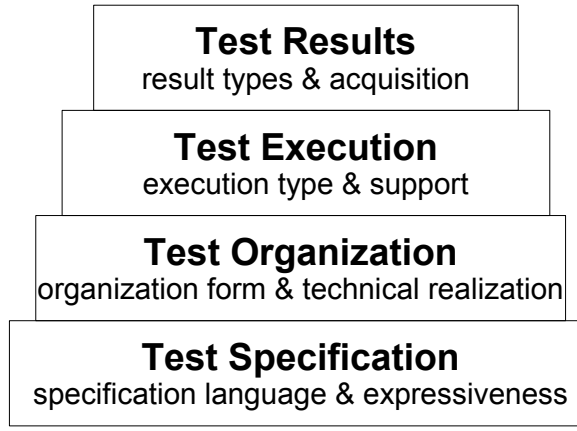

Fig. 6.3. Layers for unit testing frameworks

run, results must be gathered and presented to the user, which is done in the test results layer.

6.5.1 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 (PUT) and its partner processes.

BPEL interfaces are described using WSDL port types and operations. However, the WSDL syntax lacks a description of the actual protocol of a Web service, i.e. which operation must be invoked after or before another operation (for a discussion, see [1, 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 check whether it has been followed or not.

As for the information flow between the BPEL composition and its partner processes, the data can be differentiated between incoming and outgoing data from the perspective of the test.

The test specification must be concrete enough to validate the correctness of incoming data as well for creating outgoing data. As pointed out by [11], incoming data errors can be classified into three types:

1. incorrect content
2. no message at all, when one is expected
3. 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 extreme:

1. Data-centred approach (e.g. using fixed SOAP data, augmented with simple rules): Incoming data from the process is compared against a predefined SOAP message (which, e.g., 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 defines which replies to send. This approach is not only very simple, but also least expressive to implement tests in.
2. Logic-centred approach (e.g. 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 also created by a program. This approach is very flexible and expressive, but requires a lot more work by the test developer and is therefore more expensive to implement.

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").

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

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

1. Automation: The ultimate goal of a BPEL testing framework is repeatable automated testing. This means the test must be executable as a whole. In turn, this indicates that the test must be specified in an unambiguous, machine-readable and executable form. The more sophisticated the test logic, the more complex the test execution will be.
2. Tool support: It should be possible to automate at least some of the steps a test developer must do for creating the test specification. The effort needed to automate a test can become quite high. Consequently, it is necessary to relieve the test developer of the more tedious tasks and let him focus on the actual problem.

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

6.5.2 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 composition, it is not yet bound to a specific BPEL composition, 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 composition for testing. Additionally, it is beneficial to allow testers to organize their test cases into groups, which are called test suites in existing frameworks.

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 composition 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 composition binding again. This reduces the work done by the tester, because such bindings can become very complex.

There are two basic approaches to test organization:

1. Integrated test suite logic: The first approach is to integrate test organization with the test specification. This is possible only when a sophisticated test specification method is in place (e.g. when using a high-level language). This approach has the benefit of being very flexible for the test developer.
2. Separate test suite specification: The second approach is to allow formulation of separate test organization artefacts. These artefacts could include links to the actual test cases and the test fixture.

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

6.5.3 Test Execution

For normal execution, BPEL compositions are usually deployed into a BPEL engine, instantiated and run upon receipt of a message triggering instance creation. However, for testing a BPEL composition there are other possibilities too.

BPEL testing means executing a BPEL composition with a test environment, the so-called "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:

1. Simulated testing: Simulated testing, as defined here, means the BPEL composition is not actually deployed onto a server and invoked afterwards by 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. Simulated BPEL execution works only if the engine supports debugging, i.e. it has a rich API for controlling the execution of a BPEL instance. While most engines do support such features, unfortunately they are 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.

2. 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. 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 ([1]), 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 Sect. 6.5.1 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.

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.

6.5.4 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 (a good overview is given by [18]), 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, like test coverage, can be calculated.

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 composition 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 or debugging: APIs may be used to query the activity which is currently active. However, these APIs, if they exist, are vendor specific.
- During execution using instrumentation: Tools for other programming languages, like Cobertura for Java, are instrumenting the source code or binary files in order to being informed which statements are executed ([16]). Since the BPEL engine's only capability to communicate to the outside world are Web service calls, the notification need to be done this way. However, this approach imposes a high-performance penalty due to frequent Web service calls.
- During execution by observing external behaviour: 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 ([11]) to use log files produced by BPEL engines to extract information about the execution of a 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 in a graphical UI, or headless on a server. For all these cases, the test runners should be able to provide adequately formatted test results; e.g., 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 testing framework architecture is complete. In the next section, our design decisions for the BPELUnit framework are given.

6.6 BPELUnit

As part of our research, BPELUnit ([12]) has been developed. BPELUnit is the first step for addressing the difficulties encountered in unit testing BPEL compositions and is based on the layered architecture described in Sect. 6.5. Because its main focus is unit testing, the natural user group for BPELUnit are developers. Therefore, all technical details are readily accessible during and after test-runs, and XML is used intensively to define test cases, test parameters etc. BPELUnit is available under an open source license at `http://www.bpelunit.org`.

BPELUnit is implemented in Java. The core itself is not dependent on any presentation layer technique and therefore can be used from any build and development environment. Part of BPELUnit are

- a command-line client
- integration into ant for automatic builds
- an Eclipse plug-in for supporting development using various BPEL editors based on the Eclipse platform (Fig. 6.4).

The integration into development environments like Eclipse is important especially for developers, because switching between testing and developing is easier and quicker to do. Furthermore, assistants in the development environment can be used to quickly create tests, prepare values, generate XML fragments etc.

Various BPEL engines are supported, and new engines can be integrated by writing matching adapters. BPEL engines only need to support automatic

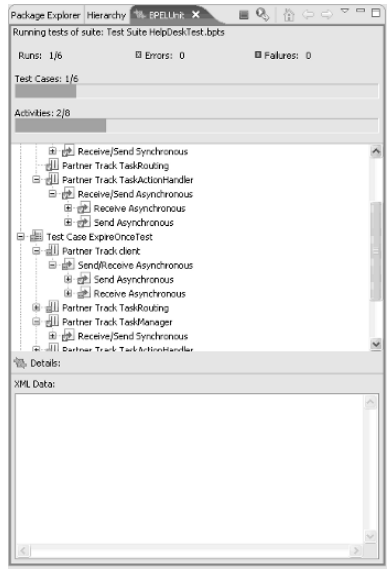

Fig. 6.4. Screenshot of BPELUnit integration into Eclipse

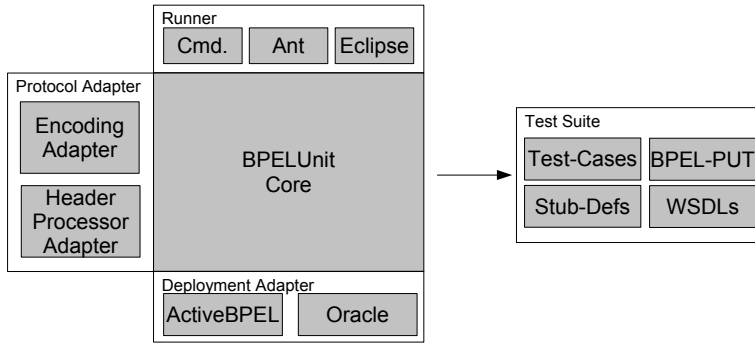

Fig. 6.5. BPELUnit architecture and test suite

deployment and undeployment of BPEL compositions. The general software architecture can be see in Fig. 6.5.

6.6.1 Architectural Layers

BPELUnit's design is aligned to the layers presented above. In the following, the design choices in each layer are described to give an overview about BPELUnit's principal architecture.

Test Specification

Tests in BPELUnit are specified in XML. The data is specified in XML, i.e. as it is in SOAP itself. Therefore, the developer has maximal control over the data sent to the BPEL composition.

The description of interactions of the BPEL composition can be verified by using XPath statements applied to the returned data. XPath is the natural choice for selecting data from XML documents. Furthermore, the interaction style between partners and composition can be stored in the specification: So far one-way (receive-only and send-only), two-way synchronous (send-receive and receive-send), and two-way asynchronous (send-receive and receive-send) are supported.

Test Organization

The test specification is organized as a set of parallel interaction threads. Each thread describes expected communication to and from one partner of the BPEL composition. These tests can be grouped into test suites. A test suite references all necessary elements for its corresponding tests: The WSDL

descriptions of services, XML schemata etc. Furthermore, the suite defines the test environment, as it contains all set-up information like the server and URLs.

In order to ease the creation of tests, test cases can be inherited: Common interaction sequences can be defined once and inherited into another test case. The new test case can add new partners and input values. This way, much effort can be saved since tests for the same compositions normally differ only slightly.

The test suites containing test cases are stored using XML. Their files normally end in .bpts (BPel Test Suite). The schema contains the following components (as illustrated in Fig. 6.6):

- A **name** used for identifying the test suite.
- The **base URL** under which the mocks should be accessible.
- Within the **deployment section** the BPEL process and all partner WSDL descriptions are referenced. The partner descriptions are used for creating the mocks.
- The **Test Cases** contain the client track responsible for feeding input to the BPEL process and the partner definitions. Those partner definitions are used to create the stubs' logic: Expected values and data to send back to the process are defined within the partner definitions.

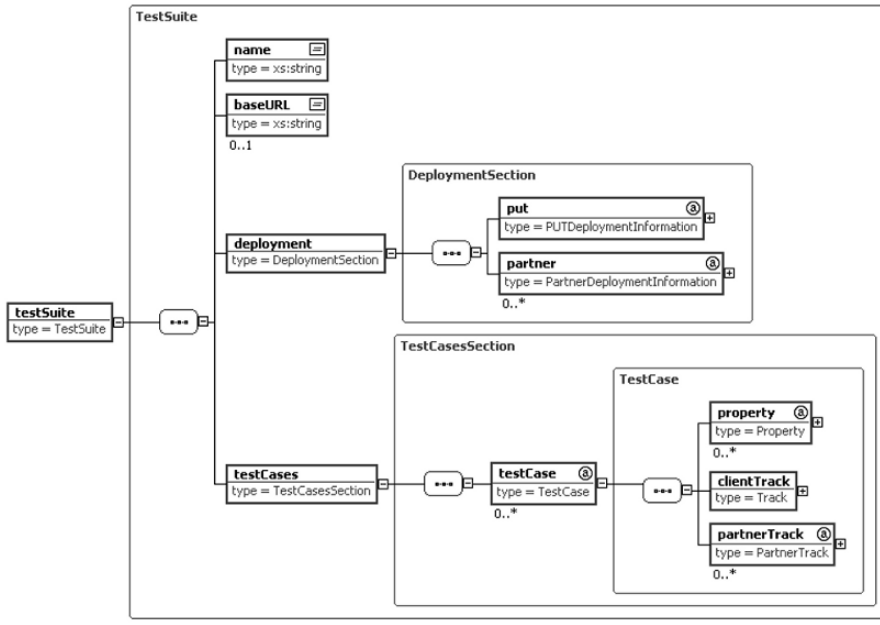

Fig. 6.6. XML schema of test Suite specifications

Test Execution

The aim of BPELUnit's execution layer is to take most of the burden from the developer. Test execution in BPELUnit can automatically deploy and undeploy compositions on servers, and offers a stub engine which resembles the behaviour specified in the parallel execution threads.

Especially, the mock engine – as it simulates partners of the BPEL composition – is quite complex. It simulates a complete Web service stack and can parse and handle the most important SOAP constructs. It contains a HTTP engine for receiving and sending SOAP messages, can process RPC/literal and document/literal styles and transparently handles callbacks using WS-Addressing. Other styles and SOAP processing options can be added by third-parties through extensions.

Test Results

Since BPELUnit uses a concrete BPEL server for execution, gathering runtime statistics is difficult. Up to now, BPELUnit only reports successful test cases, failures and errors. A failure represents an application-level defect. This normally indicates that an expected value is not received. In contrast, an error indicates a problem with the Web service stack: a server may not be responding, wrong error codes may have been sent etc.

BPELUnit itself does not offer a GUI or extensive output facilities. Instead, the test results are passed to front-ends, e.g. the Eclipse plug-in. The front-end processes and visualizes the test results.

6.6.2 Mocking Architecture

The main advantage of using BPELUnit – compared to other Web service testing tools – is its ability to emulate processes' partners. The real partners are replaced by simulated ones during run-time. The simulated partners are called stubs. At least one stub is needed per test, i.e. the partner stub. The partner stub is the partner initiating the BPEL process.

The behaviour of the stubs is configured in BPELUnit by the means of partner tracks. A partner track describes expected incoming messages and the corresponding reply messages to the process. The incoming messages can be checked by XPath statements for all relevant information, e.g. is a sent ID correct, is there certain number of products supplied etc. These checks are used by BPELUnit to evaluate whether the test was successful, i.e. if all checks were successful. Especially, the partner stub will check the final result of the BPEL process for an expected output.

Whenever a test is started, BPELUnit will start a thread for each mock. The thread is configured using the information supplied by the partner track definition. Afterwards, BPELUnit will open a port on which an own Web

service stack listens. The Web service stack decodes the incoming message, and routes the incoming requests to the matching mock. The mock consequently processes the request by checking the incoming message for validity and the correct information as expected by the partner track. Afterwards, a reply message is sent back, which is again routed through BPELUnit. Within the partner track, the tester can give definitions to assemble parts of the reply message for copying dynamic elements, like dates and times into the message, which cannot be statically defined.

The mocks do not need to deal with SOAP details, because all the SOAP-related work is done by the framework itself. An example definition of a partner track which checks the incoming message looks like this ([12], p. 71):

```
<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>
```

The client is a synchronous send and receive client, which checks whether a booking has completed successfully or not.

6.6.3 Extension of BPELUnit

BPELUnit itself is a basic implementation of a unit testing framework which handles test organization and execution well. However, the SOAP protocol and the BPEL application landscape are very complex and diverse. The SOAP protocol is very extensible, and there is no standard for accessing BPEL servers in order to deploy and undeploy processes to name two of the biggest problems.

BPELUnit supports SOAP over HTTP with literal messages for all mocks. If a process accesses other services, they cannot be mocked with BPELUnit in its current version. Besides being open source software, BPELUnit offers extension points for plugging in new protocols and header processors. Since BPELUnit itself calls processors after handling all incoming and outgoing

SOAP messages, new encodings and headers processors can be added independently of the BPELUnit source tree. For instance, the default BPELUnit distribution ships with WS-Addressing support, which is implemented as a header processor.

Another plug-in interface is offered for deployment and undeployment: Since all servers have their own way of handling deployment, it is necessary to separate these operations and make them extensible. New servers can be supported by BPELUnit by adding a corresponding plug-in. BPELUnit ships with support for ActiveBPEL, the Oracle BPEL Engine and – as a fall-back option – for manual deployment.

6.7 Example

Within this chapter the common example from the introduction is used. However, some additional technical properties will be presented at the beginning.

This example concentrates on a ticket reservation system and its associated services. The reservation system is developed and operated by a fictional, touristic company. This company is offering their services as Web services described in WSDL. For fulfilling their customers' requests, various databases of partners need to be queried: Different hotels and restaurants can be looked up and tickets can be ordered and reserved. Therefore, partner companies' services have to be integrated into the service composition.

A hotel is one of the touristic service provider's partners. It has developed a BPEL composition for fulfilling the reception of hotel reservations which includes the payment as well. The payment is realized by a Web service by a bank. The whole service architecture can be seen in Fig. 6.7.

From the point of view of the touristic service provider, the reservation services are atomic services. They can only see a black box which is outside their development organization and their control. For the hotel, however, the reservation service is a complex process using the bank's services as atomic services.

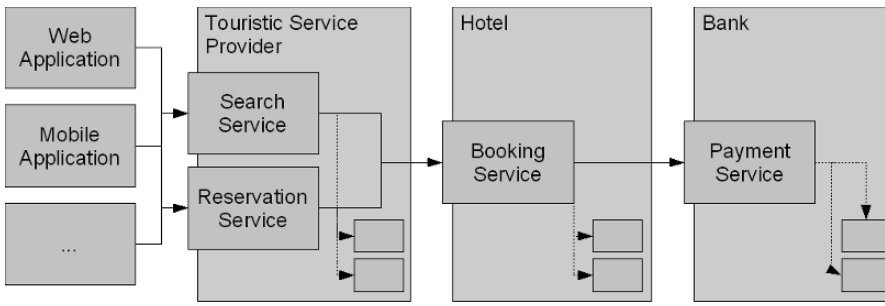

Fig. 6.7. Service architecture in the example

This scenario includes some problematic challenges for testing the corresponding compositions. Focusing on the touristic provider's process, it has no single reservation service: The Web service, which is actually called, depends on which hotel is to be booked. The provider itself has no own implementation of such a service and cannot randomly book and cancel hotel rooms for testing purposes. Therefore, the developers need to carefully unit test their process using stubs in order to minimize possible defects. For testing the whole system, organizational problems are dominant. However, the touristic provider has been able to get three hotels to offer them a test account which does not make effective bookings or cancellations. Therefore, the these parts of the system can be tested in integration tests thereby minimizing possible defects through misunderstood WSDL interfaces.

In Fig. 6.8 the touristic service provider's BPEL process is illustrated with a unit test: All hotels are queried in parallel and afterwards the results are merged. The best three results are returned. For simplicity, only a brief overview of the process is given. The unit test suite for this process simulates the hotels by returning a predefined set of possible booking options. Thereby, the correct merging of the results is validated. Moreover, service failures can be simulated by returning SOAP faults to the BPEL process to show correct behaviour in case of partner service failures. While the test could not find any errors in the merging part of the process, errors were not correctly caught in the BPEL process. This leads to termination of the whole process, instead of proceeding with only the available results. In this test, BPELUnit controls the whole execution and all mocks. The BPEL process is deployed onto the server and the test suite is run. There are no organizational borders conflicting with the test.

However, using unit tests alone, it is not possible to detect failures hidden in the communication between the BPEL process and a Web service. Web services are often created by exposing functionality written in traditional

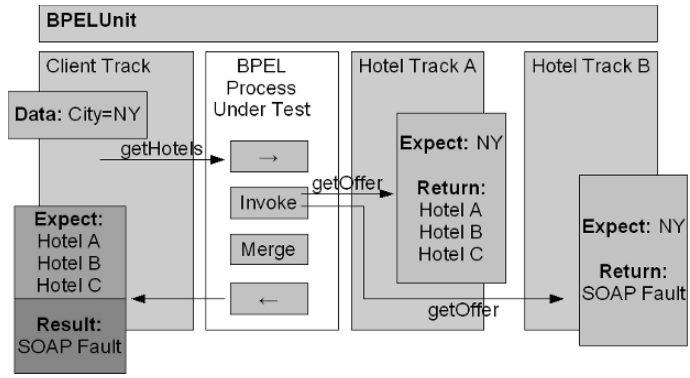

Fig. 6.8. Provider's unit test

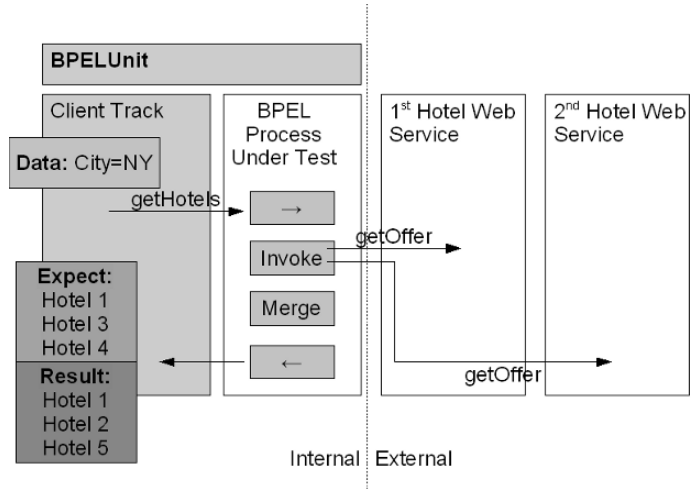

Fig. 6.9. Provider's integration test

languages. In most programming languages, indexes are counted from 0, while in XPath the first index is 1. What convention is used by a service is not stored in its WSDL description since it only defines that an integer is required. To counter such mistakes, BPELUnit can be used to do integration testing. In contrast to unit tests, no services are mocked. Instead, the original services are used. Since the touristic service provider is able to access some real services, this is possible. Such tests are consequently able to detect problems concerning the interfaces.

In the example's case, it is likely that a programmer, who misunderstood the index while writing the BPEL process, will consequently write a mock which waits for the wrong parameter. Therefore, the unit test is wrong in this regard and the error will not be spotted. However, using the test accounts during integration testing, it is possible to detect such failures as illustrated in Fig. 6.9. The BPELUnit test awaits the list of hotel offers. However, a wrong list is returned due to the wrong index. The testers can see this behaviour, report the bug and update the unit tests accordingly.

For integration tests, BPELUnit only controls the client track and the deployment of the BPEL process. However, the services used in this example are the real services. Therefore, this test spans multiple organizations.

6.8 Conclusions and Outlook

Testing service compositions, most notable such modelled in BPEL, is a relatively new aspect for quality assurance in software projects. Therefore, experiences and experience reports concerning testing are lacking. The first steps

taken to better support the testing process is to try to improve tool support: This support must address the distributed nature of services and the necessary test set-up routines for deployment of all necessary compositions and stubs.

BPELUnit is the first version of such a unit testing framework supporting BPEL. It manages test cases and contains a stub engine, which can be parameterized by the test suites. The design is extensible for adding support of further BPEL engines and front-ends.

Next aim for the future is support for gathering metrics during test execution. Unfortunately, this is a tremendous effort due to missing standards for BPEL debugging and process introspection. However, adding generic interfaces which need to be implemented by the corresponding BPEL engine adapters is possible with support for one or two BPEL engines in the standard distribution.

While the stub facility of BPELUnit is very powerful, it cannot deal with dynamic service discovery. This type of discovery poses a significant challenge for all testing activities related to SOA. Since the service to be called is determined at run-time, it is not necessary to replace the service endpoint in the deployment. Therefore, other means must be found to redirect the service call to the stub service.

Another important aspect is the parallelization of tests: At least unit tests should be independent of each other, so their execution could be distributed to different test machines and be done in parallel. For other unit testing frameworks, research concerning distribution is available, for e.g. by [10] and [17], which should be adopted for BPELUnit or comparable frameworks as well.

Furthermore, testing habits and likely defects in BPEL compositions need to be empirically studied. Interesting questions would be, e.g., in which parts of a BPEL composition errors are likely to occur, by which rate certain tests can reduce defects in the software product concerning service compositions etc.

BPELUnit can serve as a stable foundation for all these research questions. Moreover, it can be used in a production environment for finding defects in developed BPEL compositions.

References

1. Alonso, Gustavo, Casati, Fabio, Kuno, Harumi, and Machiraju, Vijay (2003). *Web Services*. Springer, 1st edition.
2. Beck, Kent (2000). *Extreme Programming Explained*. Addison-Wesley.
3. Dijkstra, Edsger Wybe (1971). *Structured programming*, chapter Notes on structured programming, pages 1–82. Academic Press.
4. Flohr, Thomas and Schneider, Thorsten (2006). Lessons learned from an XP Experiment with Students: Test-First needs more teachings. In *Proceedings of the Profes 2006*.

5. George, Boby and Williams, Laurie (2003). A Structured Experiment of Test-Driven Developmen. *Information and Software technology*, 46(5):337–342.
6. java.net (2006). WSUnit - The Web Services Testing Tool. WWW: `https://wsunit.dev.java.net/`.
7. jmock.org (2006). jMock. WWW: `http://www.jmock.org/`.
8. JUnit.org (2006). JUnit. WWW: `http://www.junit.org`.
9. Juric, Matjaz B., Kezmah, Bostjan, Hericko, Marjan, Rozman, Ivan, and Vezocnik, Ivan (2004). Java RMI, RMI tunneling and Web services comparison and performance analysis. *SIGPLAN Not.*, 39(5):58–65.
10. Kapfhammer, Gregory M. (2001). Automatically and Transparently Distributing the Execution of Regression Test Suites. In *Proceedings of the 18th International Conference on Testing Computer Software*.
11. Li, Zhongjie, Sun, Wei, Jiang, Zhong Bo, and Zhang, Xin (2005). BPEL4WS Unit Testing: Framework and Implementation. In *ICWS '05: Proceedings of the IEEE International Conference on Web Services (ICWS'05)*, pages 103–110, Washington, DC, USA. IEEE Computer Society.
12. Mayer, Philip (2006). Design and Implementation of a Framework for Testing BPEL Compositions. Master's thesis, Gottfried Wilhelm Leibniz Unversität Hannover.
13. Mayer, Philip and Lübke, Daniel (2006). Towards a BPEL unit testing framework. In *TAV-WEB '06: Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications*, pages 33–42, New York, NY, USA. ACM Press.
14. McConnell, Steve (2004). *Code Complete*. Microsoft Press, 2nd edition.
15. Myers, Glenford J. (1979). *The Art of Software Testing*. John Wiley & Sons.
16. Project, Cobertura (2006). Cobertura Homepage. WWW: `http://cobertura.sourceforge.net/`.
17. Safi, Bassim Aziz (2005). Distributed JUnit. Bachelor Thesis at University Hannover.
18. Zhu, Hong, Hall, Patrick A. V., and May, John H. R. (1997). Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427.