BPEL Conformance in Open Source Engines

Simon Harrer, Jörg Lenhard and Guido Wirtz
Distributed Systems Group
University of Bamberg
Bamberg, Germany
{*simon.harrer,joerg.lenhard,guido.wirtz*}*@uni-bamberg.de*

***Abstract*—More than five years have passed since the final release of the long-desired OASIS standard of a process language for Web Services orchestration, the *Web Services Business Process Execution Language* (BPEL). The aim of this standard was to establish a universally accepted Web Services orchestration language that forms a core part of service-oriented architectures and, because of standardization, avoids vendor lock-in. By now, several fully conformant engines should have arrived in the market. It is our aim to shed light on this situation and to provide a comprehensive picture of the current state of BPEL support. We present an evaluation of the standard conformance of five open source BPEL engines. To obtain these results we have developed *betsy*, a tool that allows for a fully-automatic standard conformance testing of BPEL engines. The results demonstrate that full standard conformance in contemporary engines is still far from given.**

***Keywords*-BPEL, engine, conformance testing, orchestration**

I. Introduction

The BPEL specification [1] defines a language that can be used to build stateful Web Services that take part in long-running interactions. It allows to define control- and data-flow dependencies between the invocation of other services in a process-like manner, thereby *orchestrating* the services [2]. Process definitions built in the language can be executed on an *engine* that implements the specification. Being an open standard, BPEL is intended to provide portability of process definitions among multiple engines. BPEL is widely accepted as a core part of Web Services-based service-oriented architectures (SOAs) [3].

An area where BPEL typically is of relevance is service-based business-to-business integration. Here, it is necessary to integrate processes that are executed by several autonomous partners [4]. Scientific approaches try to tackle this problem with a combination of choreography and orchestration models [2]. A choreography model specifies a communication protocol between the different partners from a global point of view, an orchestration model, defined in BPEL, implements a partner-local process. The overall protocol can be implemented by translating the global model into several local orchestrations (top-down), see for example [5], [6], [7], [8], or by constructing the global protocol from preexisting orchestrations (bottom-up), as in [9]. All these approaches make heavy use of BPEL and implicitly assume that fully conformant engines will be available one day. If this premise is not fulfilled, they unleash their full potential and benefit only in theory. Standard-conformance in BPEL engines is also a critical enabler for the portability of process models and the avoidance of vendor lock-in, two of BPEL's main promises.

In this paper, we investigate the degree of today's standard conformance for BPEL. Specifically, we investigate the market of open source BPEL engines, namely the five engines *Apache ODE*, *bpel-g*, *OpenESB*, *Orchestra*, and *Petals ESB*. To assess their standard conformance, we have implemented the tool *betsy*. Betsy is open source and documented in [10]. This tool comprises a BPEL conformance testing suite, leveraging a set of almost 130 process definitions that define conformant behavior, and allows for the fully automatic assessment of the standard conformance of BPEL engines. Thereby, we are able to measure, in a fashion that is comparable among multiple engines, the degree of standard conformance provided by an engine and can determine which parts of the standard are not implemented. When selecting an engine, practitioners can consider these results together with other crucial characteristics, such as scalability and performance properties, for making their choice.

This assessment shows that there are several major shortcomings in the BPEL support for all engines under test. As a consequence, process definitions are tightly coupled to their execution platform and porting such definitions from one engine to another can be considered a daunting task. For the scientific approaches (cf. [5], [6], [7], [8]) it implies that they are also linked to the platform for which they have been tested and cannot, generally, be assumed to work on any platform.

The next section discusses related approaches and delineates our work from other approaches on testing BPEL. Thereafter, we explain the environment and structure of our testing setup. Sec. IV presents and discusses our results. Sec. V summarizes the paper and proposes future work.

II. Related Work

The testing of BPEL has received some attention so far[1], and concentrates on (i) *unit testing* of executable BPEL

[1]A comprehensive overview of academic approaches to web service testing is given in [11]. A subset of these approaches applies to BPEL.

processes, (ii) *conformance checking* of BPEL processes to a certain specification or formalism, and (iii) *performance testing* of BPEL processes or engines. The testing of the standard conformance of BPEL engines is still an open question, however.

Basically, our work builds on unit testing approaches and shares the aspect of testbed generation with performance testing approaches. It differs from conformance checking approaches in the system under test, being the middleware in our case and concrete process models for other approaches. To the best of our knowledge, no other conformance assessment of BPEL middleware can be found yet.

The area of unit testing BPEL processes is one that received considerable interest, but even here more work is called for [12]. In this area, the *BPELUnit* project [13] is most widely accepted. BPELUnit allows for the construction of unit and integration tests for BPEL processes that run on specific engines. The aim of BPELUnit, and related unit testing approaches for BPEL, is to test and verify the correctness of specific BPEL processes. Here, we aim at the testing the conformance of BPEL engines; that is, the systems under test are different. Our tool is similar to BPELUnit in the manner that it allows for the automatic deployment of BPEL processes and execution of test cases for these processes for specific engines. In fact, betsy internally uses a unit testing framework, *soapUI*[2], to automate the test execution and reporting, and builds its conformance testing workflow on top of that.

Conformance checking of BPEL is generally not understood as the testing of the conformance of a BPEL engine to the BPEL specification, but refers to the verification of the behavioral properties of a concrete BPEL process. For instance, it is verified that a concrete BPEL process behaves as specified by (conforms to) an abstract process model. Examples of approaches using this type of conformance checking are [14], [15], [16], [17], [18]. Here, we do not focus on approaches for verifying behavioral conformance of concrete process models, but instead on implementation conformance of the middleware to the standard specification in the sense of [19, pp. 203-208],[20]. That is, a conformant implementation of the BPEL standard is an implementation that satisfies both, static and dynamic, conformance requirements explicitly defined in the specification [1].

The main approaches in the area of performance testing of BPEL engines are *SOABench* [21] and *GENESIS2* [22]. Both essentially are testbed environments that can be used to generate testbeds for complex service-oriented systems. Whereas GENESIS2 is directed at service-oriented systems in general, SOABench is specifically aimed at the testing and analysis of the performance characteristics of BPEL engines. Each tool defines a domain model to automatically generate and execute test cases and provides a plugin mechanism to extend the execution environment with new engines. As we address BPEL, our domain model has a larger intersection with that of SOABench than with GENESIS2. However, SOABench's model is more complex than that of betsy which is necessary to get a more fine-grained control of the testing environment. This in turn is a prerequisite for gathering performance metrics. As betsy is directed at conformance testing and not performance testing, it has no such requirements. Finally, whereas SOABench comes with four BPEL process definitions that are aimed at testing the performance and scalability of an engine, betsy comes with a set of almost 130 processes that have the aim to assess the standard conformance of an engine. Moreover, betsy natively supports five engines instead of three.

A completely different approach of achieving BPEL conformance is based on formalizing the operational semantics and deriving an implementation of these semantics. This idea and the semantics are outlined in [23].

To sum up, our work builds on unit testing and testbed generation approaches for BPEL and adds the layer of conformance assessment on top. To date, no such comprehensive conformance evaluation can be found. Our tool implementing the conformance assessment is open source and documented in [10].

III. Testing Setup and Environment

Our testing setup consists of a testing tool that is capable of installing, deleting and communicating with a variety of BPEL engines, and a suite of engine-independent conformance test cases. The test engine instruments these test cases to produce engine-specific deployment artifacts for every test case and executes these artifacts for all engines. To assure the quality of the test cases, they have been validated using the XSD files from the BPEL 2.0 specification, have been peer–reviewed in our group, and are publicly available for scrutiny and improvement. Finally 95% of the tests are passed by at least one engine which indicates that the process definitions are correct. This section describes the engines under test, the tool, and the structure of the test cases.

A. Engines under Test

All engines we investigate are open source and freely available. They are rated as mature projects by their respective distributors and are actively maintained.

Apache ODE: Today, Apache ODE[3] is the most well-known and most widely used open source BPEL engine. It is incorporated in various open source enterprise services buses (ESBs). The revision used in the tests is ODE 1.3.5.

bpel-g: The bpel-g engine is a derivate of the former ActiveBPEL by Active Endpoints. Whilst ActiveBPEL is no longer provided, bpel-g is still under development as a

[2]soapUI is a unit testing framework not specifically attached to BPEL, but to Web Services in general. For more information, see http://www.soapui.org/.

[3]The project page is available at http://ode.apache.org/.

Google Code project[4]. The engine comprises the functionality provided by ActiveBPEL. Our work uses the 5.3 snapshot of bpel-g, being the most recent version available at the time.

OpenESB: The OpenESB is an open source ESB that includes a BPEL engine. It is was maintained by Sun preceding Sun's acquisition by Oracle. OpenESB is commonly collocated with the Glassfish application server to form a full enterprise integration solution. The project homepage is http://openesb-dev.org/ and the version used here is 2.2.

Orchestra: Orchestra is developed by the OW2 consortium and available at http://orchestra.ow2.org/. It executes BPEL processes on a generic process virtual machine. We analyze Orchestra 4.9, being the most recent stable revision at the time of writing.

Petals ESB Petals ESB is an open source ESB that includes a BPEL engine. It is available at http://petals.ow2.org/ and is developed by the OW2 consortium, just as Orchestra. Instead of reusing Orchestra as a BPEL engine, Petals ESB provides a separate engine, namely EasyBPEL[5]. In the tests, we use EasyBPEL 4.0.

B. The Tool betsy

The testing engine derives its name, *betsy*[6], from **B**PEL **E**ngine **T**est **Sy**stem. It is open source, developed in Groovy, and makes extensive use of *Ant* and soapUI. It comes with a domain model for representing test cases and engines and executes these test cases in a conformance testing workflow. This section sketches the domain model and the testing workflow.

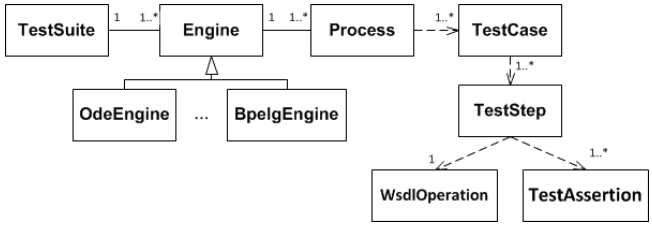

Figure 1. Betsy's Domain Model

The domain model is depicted as a UML class diagram in figure 1. The key elements are the `Process` and the `Engine` classes. A `Process` references engine independent BPEL, WSDL, XSD and XSLT definitions that encapsulate a single feature of BPEL. This is the basis for a single conformance test. An `Engine` defines methods for managing a concrete engine instance and for deploying process definitions to that instance. This class has to be extended for each engine under test. The tool can be extended to support additional engines using solely this class. For each process, a set of `TestCases`, consisting of several `TestSteps` that are verified through `TestAssertions` can be defined. A `TestCase` verifies that a sequence of synchronous or asynchronous message exchanges (`TestSteps` and `WsdlOperations`) is correct for a specific BPEL `Process`. On execution, betsy links all `Processes` with all `Engines` in a `TestSuite` and executes all resulting tests sequentially. A `TestCase` passes if all its `TestSteps` have passed and a `Process` (e.g. a feature of BPEL) is supported if all its `TestCases` have passed. If at least one, but not all, `TestCases` fail, the feature is partially supported. The correctness of a `TestStep` is asserted by evaluating the content of response messages, if there are any. Altogether, betsy comes with almost 130 `Processes` and related `TestCases`.

The testing workflow is depicted in figure 2. The workflow divides into *setup*, *tear down* and *test execution* phases. `Prepare Folders` marks the setup phase and `Generate Reports` the tear down phase and both are being executed exactly once, before and after the test execution phase, respectively. The test execution phase takes place per engine and per process (per test case) and is repeated strictly sequential. This avoids any side-effects that could occur through parallel tests. Due to the number of engines and processes, it is repeated approximately 650 times.

The setup phase prepares the file system for the upcoming test run, deleting and rebuilding the required folder structure. The test execution phase divides into a sequence of substeps. First, the engine independent BPEL, WSDL, XSD and XSLT files are transformed into engine specific files and are packaged into a deployable archive (`Generate BPEL`). Additionally, deployment descriptors are generated based on the BPEL and WSDL files using XSL transformations and are included within the deployment archive. Second, each `TestCase` is transformed into a corresponding soapUI project configuration (`Generate Test`). Third, the engine that is to be tested is installed and started (`Install Engine` and `Start Engine`). This involves the deletion of a previously installed instance of the engine, followed by its download (if needed), extraction and configuration. The complete reinstallation of an engine for each test ensures that there are no side-effects on a test through a previous test. Fourth, the previously created deployment archive is deployed onto the engine (`Deploy BPEL`). This is achieved via hot deployment or API calls, depending on the engine under test. Fifth comes the actual test execution; that is the transmission of SOAP messages to the processes' endpoint and the evaluation of the responses (`Execute Test`). The message transmission is performed by soapUI by executing the project configurations defined in step two of the test execution phase. The test results and messages exchanged are stored for later evaluation. Last, the engine is stopped and the test execution phase is repeated for the next test. After all tests have been executed, the tear down

[4]The project can be found at http://code.google.com/p/bpel-g/.

[5]Its documentation can be found at http://research.petalslink.org/display/easybpel/EasyBPEL+Overview. For brevity, in the rest of the paper we refer to this engine as Petals ESB.

[6]The project homepage is https://github.com/uniba-dsg/betsy/tree/soca-2012. Betsy's functioning is documented in a technical report [10].

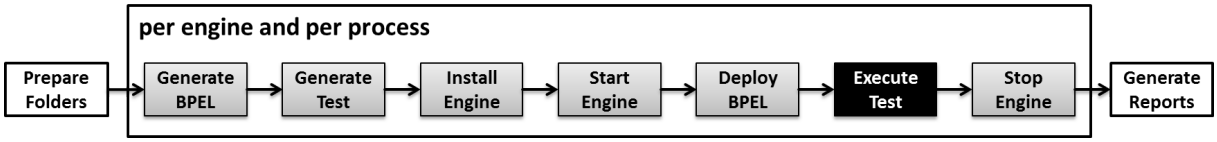

Figure 2. Test Process Workflow.

phase (`Generate Reports`) takes place. In this phase, the results are aggregated and HTML reports visualizing them are generated. This enables the user to drill down from a high-level result overview to the detailed messages exchanged by engine and process for each test.

C. Conformance Test Cases

This section provides a more detailed outline of the structure and configuration of the test cases used to assess BPEL standard conformance. Each conformance test case describes a specific feature of BPEL in relative isolation. A test case is subdivided into a test case definition and a test case configuration. The definition comprises BPEL, WSDL, XSD and XSLT files. The configuration consists of instances of `Processes` and `TestCases` of the domain model (cf. section III-B) referencing the files of a test case definition.

The test cases are derived from the requirements defined in the BPEL specification [1] using the notational conventions [24] (i.e. MUST, MUST NOT, etc.). Specifically, we provide multiple process definitions to test every activity, attribute and almost every fault that is part of executable BPEL. All test cases are classified into three groups:

basic-activities: This group contains test cases for every basic activity of BPEL [1, pp. 84–97]. This includes the `invoke`, `receive`, `reply`, `assign`, `throw`, `wait`, `empty`, `exit`, `validate`, and `rethrow` activities, as well as faults related to them.

structured-activities: The second group comprises structured activities [1, pp. 98–114]. This includes `sequence`, `if`, `while`, `repeatUntil`, `pick`, `flow`, and `forEach` activities. Again, faults related to these activities belong to this group as well.

scopes: Although being structured, `scopes` [1, pp. 115-147] are treated separately. The last group contains tests for `scopes`, `fault-`, `compensation-`, `termination-`, and `eventHandlers`. Furthermore, the scope-local definition of `variables`, `partnerLinks`, `messageExchanges` and `correlationSets` is also investigated here.

Every test definition implements the same WSDL interface to enable a unified handling of tests. The interface is intended to be as simplistic as possible, to ensure that all engines support it, whilst being complex enough to test all features of BPEL. It contains a `partnerLinkType` and several message definitions, with all messages containing a single message part of the type integer. Thereby, we avoid problems that result from the processing of large documents, as it is not our intention to assess XML processing capabilities here. The `portType` is made up of two operations, namely (i) a synchronous one that may also reply with a fault and (ii) an asynchronous one. The binding for these operations is the most basic and plain one available, thereby having a high probability of being supported by every engine: document/literal style over HTTP [25, Sec.3]. We also provide a similarly structured WSDL definition for, and an implementation of, a partner service that is required to test `invoke` activities.

The tests for a specific feature of BPEL are not strictly isolated, which is also no requirement for conformance tests [19, pp. 203-208]. Some features are not testable in isolation, such as `faultHandlers` that require a fault to be thrown in the first place. Furthermore, to verify the correctness of a test, it is necessary to have an output available that can be evaluated. Consequently, all process definitions we use as conformance tests contain certain elements and most do contain synchronous operations. Listing 1 outlines the general structure that applies to most tests. The activities therein with their specific configuration could be verified to be supported by all engines, so they do not influence the results of other tests.

Listing 1. Outline of the Process Definitions

```
<process>
   <partnerLinks />
   <variables />
   <sequence>
        <receive />
        <!--Test implementation-->
        <assign />
        <reply />
        <!--More test implementation, if message
             exchanges are involved-->
   </sequence>
</process>
```

All the engine independent files described before are referenced in a test configuration. This configuration is built according to our domain model (cf. section III-B); that is, a `Process` links to all the files for a given test.

Three areas required for executable BPEL are not completely specified and remain a design-choice for an implementor of the standard. This is the exact structure of a partner reference, necessary for the assignment of `partnerLinks`, the URI scheme used to identify XSL stylesheets and the behavior of the engine if a fault is propagated to, and not handled by, the root-level scope of a

process that still has open request-response interactions. We decided to use WS-Addressing `EndpointReferences` [26] (encapsulated in BPEL's `service-ref` container) as partner references and identify XSL resources by their filename. This implies that an engine that supports dynamic binding, but not with `EndpointReferences`, will fail our test case. Concerning fault propagation we test for the mechanism applied by basically any high-level programming language, such as Java or C#, which is also a prerequisite for distributed fault handling [27]: We expect an uncaught fault at root-level to be forwarded to the recipients in open request-response operations.

IV. Test Results

The results listed here originate from a run of the complete test set for all engines[7]. The execution of this run took approximately 10 hours. Figure 3 provides an overall picture of the run. The results for every engine are given either in total ($\sum$) or subdivided in the three test case groups basic activities (BA), scopes (S), and structured activities (SA). Each column contains the amount of successful, partially successful, and failed tests. An overview of the amount of successful test cases per activity and engine is given in table I. A detailed description of every single test case can be found in our technical report [10].

As depicted in figure 3, the engine with the highest amount of successful tests, and, therefore, ranking highest when compared to the other engines, is bpel-g. Apache ODE and OpenESB are ranked second and third with a number of 86 and 84 successful tests, respectively. Orchestra comes fourth, passing less than half of the total amount of tests, followed by Petals ESB which passes only about one quarter.

Still, bpel-g fails 20% of the total amount of test cases followed by 33% and 34% for Apache ODE and OpenESB, respectively. The fact that every engine fails a double-digit percentage of the test cases means that no engine is nearly completely conformant to the standard. Except for Petals ESB, the highest degree of failed tests can be diagnosed for basic activities as shown in figure 3.

A. Engines

In this section, we briefly discuss important findings for each of the engines.

Results for bpel-g: As previously stated, bpel-g has the highest number of successful tests and consequently the highest conformance rating. In comparison to the other engines, it especially stands out in fault handling; that is, the faults that are expected to be thrown in a given situation, are also thrown by bpel-g. Still, bpel-g's fault handling is not complete. Furthermore, bpel-g is the only engine to support dynamic partner binding with WS-Adressing `EndpointReferences` (i.e. it is the only engine that was able to assign an `EndpointReference` to a `partnerLink` and afterwards invoke the new endpoint correctly). Its main disadvantage is the lack of support for timing activities. The use of `xs:dateTime` and `xs:duration` in `until` or `for` elements is not supported and `wait` and `onAlarm` activities do never complete. Moreover, `terminationHandlers` are not supported as well. It has, however, the highest support for structured activities failing only four tests in this group.

Results for Apache ODE: Apache ODE is positioned second in the ranking of the different engines. Many process definitions are not deployed, because ODE detects the use of activities that are known to be unsupported in this engine. For instance, this applies to process definitions that use `toParts` or `fromParts` elements, or the `doXslTransform` XPath function. In-process validation, using either the `validate` activity or the `validate` attribute in the `assign` activity, is unsupported. Furthermore, any `terminationHandlers` are ignored, too.

Results for OpenESB: OpenESB ranks third, but close up to rank two. It excels in the test cases of the scope group where it provides the highest degree of support. For the other two groups, however, some deficiencies can be found. OpenESB ignores links within `flow` activities. Instead, OpenESB executes the activities in a `flow` in the order of their definition. Furthermore, the `parallel` attribute of the `forEach` activity and `to-` and `fromParts` are not supported. In contrast to nearly all other engines, OpenESB does support `terminationHandlers`.

Results for Orchestra: With the amount of failed tests exceeding the amount of successful ones, Orchestra is positioned fourth. As can be seen in figure 3, Orchestra's support for structured activities is nevertheless comparable to that of Apache ODE and OpenESB. However, concerning the tests for basic activities and scopes, Orchestra falls behind. One reason for this is Orchestra's fault handling strategy, an aspect where it differs from all other engines under test. Orchestra never propagates faults to the caller in a request-response operation, unless this is explicitly requested in a `reply` activity. If a process instance does not handle a fault and there are still open request-response operations, Orchestra replies with an HTTP 200 OK status code. This leaves the caller clueless about the fact that a fault occurred or which fault occurred, which severely hampers distributed fault handling [27]. For this reason, Orchestra also fails all tests for the `throw` and `rethrow` activities (see table I). Moreover, the `forEach` and the `validate` activity are not supported at all. All test cases for the combination of message correlation and asynchronous operations fail. Nevertheless, Orchestra is one of the two engines supporting `terminationHandlers`.

Results for Petals ESB: The last place in the ranking of the engines is taken by Petals ESB. It fails more than 75% of the test cases which indicates only min-

[7]The resulting data set can be downloaded at https://github.com/downloads/uniba-dsg/betsy/test-results-soca-2012.zip.

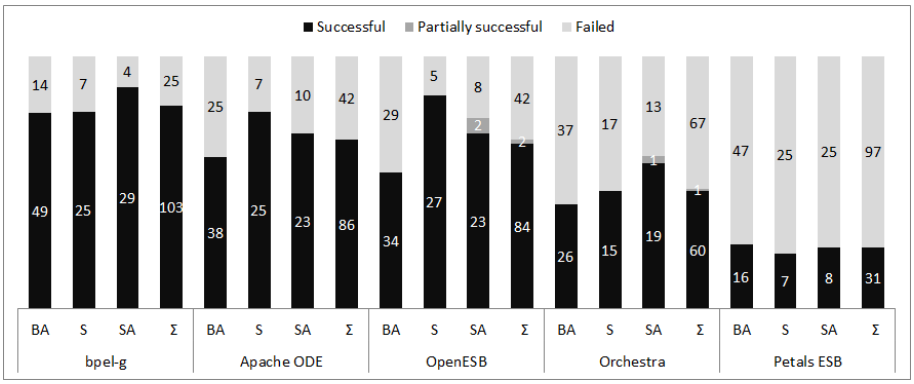

Figure 3. The number of successful, partially successful, and failed tests per engine and activity group.

Table I
NUMBER OF SUCCESSFUL TEST CASES OF EVERY BPEL ACTIVITY FOR EACH ENGINE

Group	Activity	bpel-g	Apache ODE	OpenESB	Orchestra	Petals ESB	total tests
basic activities	Assign	13	10	11	8	6	18
	Empty	1	1	1	1	1	1
	Exit	1	1	1	1	1	1
	Invoke	8	6	3	7	4	11
	Receive	4	3	1	1	1	5
	ReceiveReply	8	5	5	5	1	11
	Rethrow	3	2	1	0	0	3
	Throw	5	5	4	0	0	5
	Validate	2	0	2	0	0	2
	Variables	3	2	2	1	1	3
	Wait	1	3	3	2	1	3
scopes	Compensation	5	4	5	2	0	5
	CorrelationSets	2	2	1	0	0	2
	EventHandlers	3	7	6	4	0	8
	FaultHandlers	6	6	6	2	5	6
	MessageExchanges	3	1	1	1	0	3
	PartnerLinks	1	1	1	1	0	1
	Scope-Attributes	3	2	3	1	2	3
	TerminationHandlers	0	0	2	2	0	2
	Variables	2	2	2	2	0	2
structured activities	Flow	9	9	2	7	0	9
	ForEach	8	2	9	0	1	10
	If	5	4	4	4	4	5
	Pick	3	5	4	4	1	5
	RepeatUntil	2	1	2	2	0	2
	Sequence	1	1	1	1	1	1
	While	1	1	1	1	1	1

imalistic support. As shown in Fig 3, for no activity group the amount of successful tests is higher than 25% of the total amount of tests for this group. Looking at table I, 12 activities and features are not supported at all. These are the `flow`, `repeatUntil`, `validate`, `throw` and `rethrow` activities. On scope-level, the definition of `correlationSets`, `eventHandlers`, `messageExchanges`, `partnerLinks`, `variables`, and `compensation`-, and `terminationHandlers` are not supported. What is more, Petals ESB did not pass a single test that involves message correlation.

B. Implications

With a close look at the results in table I, some patterns can be identified. Every engine supports basic control-flow constructs, in particular the `sequence`, `if`, and `while` activities. Furthermore, basic facilities for enabling message exchanges (`invoke`, `receive`, and `reply` activities) are available in their simplest configuration along with support for data handling using the `from` and `to` or `literal` syntax in the `assign` activity. Moreover, the `empty` and `exit` activities and basic support for fault handling using `faultHandlers`, as long as either a fault is caught by its name or all faults are caught, is in place. More advanced features, such as graph-based control-flow definition, asynchronous messaging and message correlation, concurrency, compensation, or validation are far from ubiquitous. Process definitions using these features are not portable among all engines.

For instance, links are supported by only 3 out of 5 engines. Links are required to implement control-structures forming directed acyclic graphs, and are the only way for graph-based control-flow definition in the language. Graph-based control-flow definition represents an important enhancement that is extensively used in approaches for the transformation of higher-level process models to BPEL[8].

Moreover, asynchronous messaging and message correlation is a crucial enabler for the construction of long-running processes. All five engines do support asynchronous messaging. But only Apache ODE and bpel-g are able to combine asynchronous messaging with message correlation. Regarding synchronous messaging, the results show similar results as only Apache ODE and bpel-g fully support correlation with synchronous messaging while OpenESB supports this feature partially. Orchestra is only able to correlate messages sent with asynchronous operations. Lastly, message validation and `terminationHandlers` are only implemented in two engines.

Considering the fact that BPEL is a completed standard for more than five years, a higher degree of standard conformance in contemporary engines could have been expected. No single engine implements the complete (or nearly complete) BPEL specification. Even the engine that passes the highest amount of tests (bpel-g), fails about 20% of the test cases.

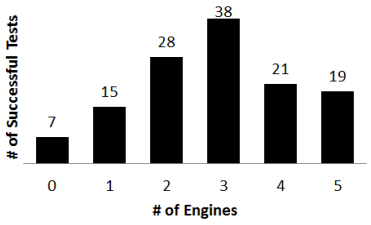

Figure 4. Number of tests in correlation to the number of engines which support them.

Incomplete implementations of the standard are not per se a thread to the portability of the process definitions running on those engines, as long as the engines implement *the same* part of the standard. However, if each engine implements a varying part, a loss of portability will be the likely result. Figure 4 displays the number of tests correlated to the number of engines that support a test. Only 19 of the 128 tests (15%) are commonly passed by all five engines, whereas seven tests (5%) are failed by all engines. The 19 successful tests assess the features described in the beginning of this section. The seven tests that are failed by all engines have been extensively peer–reviewed to reduce the probability of human error in the process definitions. As 95% of the tests work on at least one engine, the overall test suite can be considered to be valid.

[8]See for example [28] and the BPMN 2.0 standard [29] which uses links for its transformation of BPMN to BPEL.

Table II gives a pairwise comparison of the amount of successful tests shared by the engines. The number of test cases supported by two engines ranges between 75 and 24. Although bpel-g, Apache ODE and OpenESB pass 103, 86 and 84 tests respectively, the amount of pairwisely shared successful tests is considerably lower, at 75, 70 and 65. This is between 59% and 52% of the total number of test cases. With the amount of commonly implemented features ranging at this niveau, portability of process definitions is hard to achieve among the engines.

	bpel-g	ODE	OpenESB	Orchestra	Petals ESB
bpel-g	-	75	70	48	28
ODE	75	-	67	48	27
OpenESB	70	67	-	44	27
Orchestra	48	48	44	-	24
Petals ESB	28	27	27	24	-

Table II
COMMON SUCCESSFUL TEST CASES PER ENGINE PAIR

V. CONCLUSION AND FUTURE WORK

In this paper we have examined the level of standard conformance for the BPEL specification of the five open source engines bpel-g, Apache ODE, OpenESB, Orchestra, and Petals ESB. To meet this aim, we have developed the tool betsy, a software for fully-automatic and comparable standard conformance testing of BPEL engines. The test set with almost 130 test cases covers the executable part of BPEL.

The results demonstrate that the degree of BPEL conformance is far from being exhaustive, despite the fact that the BPEL specification has been finalized in 2007. BPEL conformance ranges from 80% to 24% for the different engines. Even the two engines with the highest amount of successful tests, share only 59% of the test set and the top three engines solely support the same 45% of the BPEL specification. Consequently, the advantage of portability of process definitions promised by the usage of the open standard BPEL can be called into question.

Future work comprises three main aspects: (i) extending the test set for assessing BPEL standard conformance, (ii) taking WS-* technologies into account and (iii) increasing the number of engines under test. The static analysis features of the BPEL specification [1, pp. 194–205] describe aspects that must lead to the rejection of a process definition. These features are not tested for with the current test set, which focuses solely on support for correct process models and not the rejection of faulty ones. They could be included as separate test cases which assert that a faulty process definition is not deployed. In addition, betsy could use complex real world BPEL processes as activity integration tests. This could reveal portability issues concerning the combination of activities. Moreover, service orchestration is

only a part of what is required in today's Web Services-based SOAs. Other standards, such as WS-Security or WS-ReliableMessaging, are needed for secure and reliable interaction. The conformance testing of contemporary middleware for these standards would be beneficial. Here, our tool forms a suitable basis for extension. In this paper, we have focused on open source engines. Still, there are several commercial BPEL engines available, for instance Oracle BPEL Process Manager, IBM Business Process Manager, and ActiveVOS. The assessment of these engines, and the comparison between open source and commercial engines, could also provide valuable insights.

REFERENCES

[1] OASIS, *Web Services Business Process Execution Language*, April 2007, v2.0.

[2] C. Peltz, "Web Services Orchestration and Choreography," *IEEE Computer*, vol. 36, no. 10, pp. 46–52, October 2003.

[3] M. P. Papazoglou and D. Georgakopoulos, "Service-oriented Computing," *Communications of the ACM*, vol. 46, no. 10, pp. 24–28, October 2003.

[4] C. Schroth, T. Janner, and V. Hoyer, "Strategies for Cross-Organizational Service Composition," in *MCETECH*, Montreal, Canada, January 2008, pp. 93–103.

[5] A. Schönberger, "The CHORCH B2Bi Approach: Performing ebBP Choreographies as Distributed BPEL Orchestrations," in *SC4B2B*, Miami, Florida, USA, July 2010.

[6] B. Hofreiter and C. Huemer, "A Model-driven Top-down Approach to Inter-organizational Systems: From Global Choreography Models to Executable BPEL," in *IEEE Joint Conference CEC and EEE*, Washington, D.C., USA, 2008.

[7] I. Weber, J. Haller, and J. Mulle, "Automated Derivation of Executable Business Processes from Choreographies in Virtual Organisations," *IJBPIM*, vol. 3, pp. 85–95, 2008.

[8] S. Harrer, A. Schönberger, and G. Wirtz, "A Model-Driven Approach for Monitoring ebBP BusinessTransactions," in *SERVICES2011*, Washington, D.C., USA, July 2011.

[9] G. Decker, O. Kopp, F. Leymann, and M. Weske, "Interacting Services: From Specification to Execution," *Data & Knowledge Engineering, Elsevier*, vol. 68, no. 10, pp. 946–972, 2009.

[10] S. Harrer and J. Lenhard, "Betsy – A BPEL Engine Test System," Otto-Friedrich Universität Bamberg, Bamberger Beiträge zur WIAI 90, July 2012.

[11] M. Bozkurt, M. Harman, and Y. Hassoun, "Testing & Verification In Service-Oriented Architecture: A Survey," *Software Testing, Verificaton and Reliability*, 2012.

[12] Z. Zakaria, R. Atan, A. Ghani, and N. Sani, "Unit Testing Approaches for BPEL: A Systematic Review," in *APSEC*, Penang, Malaysia, December 2009, pp. 316–322.

[13] D. Lübke, "Unit Testing BPEL Compositions," in *Test and Analysis of Service-oriented Systems*. Springer, 2007, pp. 149–171, ISBN 978-3540729112.

[14] W. M. P. van der Aalst, N. Lohmann, P. Massuthe, C. Stahl, and K. Wolf, "From Public Views to Private Views - Correctness-by-Design for Services," in *WS-FM*, Brisbane, Australia, September 2007, pp. 139–153.

[15] M. Geiger, A. Schönberger, and G. Wirtz, "Towards Automated Conformance Checking of ebBP-ST Choreographies and Corresponding WS-BPEL Based Orchestrations," in *SEKE*, Miami, Florida, USA, July 2011.

[16] W. M. van der Aalst, M. Dumas, C. Ouyang, A. Rozinat, and E. Verbeek, "Conformance Checking of Service Behavior," *TOIT*, vol. 8, no. 3, pp. 13:1–13:30, May 2008.

[17] A. Both and W. Zimmermann, "Automatic Protocol Conformance Checking of Recursive and Parallel BPEL Systems," in *ECOWS*, Dublin, Ireland, November 2008, pp. 81–91.

[18] J. García-Fanjul and J. T. Claudio de la Riva, "Generation of Conformance Test Suites for Compositions of Web Services Using Model Checking," in *Testing: Academic and Industrial Conference – Practice And Research Techniques*. Windsor, United Kingdom: IEEE, August 2006.

[19] A. P. Mathur, *Foundations of Software Testing*. Dorling Kindersley, 2009, ISBN-13: 978-81-317-1660-1.

[20] *ISO/IEC 9646-1:1994 – Information technology – Open Systems Interconnection – Conformance testing methodology and framework – Part 1: General concepts*, ISO, 1994.

[21] D. Bianculli, W. Binder, and M. L. Drago, "Automated Performance Assessment for Service-Oriented Middleware: a Case Study on BPEL engines," in *WWW*, Raleigh, North Carolina, USA, April 2010, pp. 141–150.

[22] L. Juszczyk and S. Dustdar, "Script-based Generation of Dynamic Testbeds for SOA," in *ICWS*, Miami, Florida, USA, July 2010.

[23] D. Sun, Y. Zhao, H. Zeng, and D. Ma, "An Operational Semantics of WS-BPEL based on Abstract BPEL Machine," in *SOCA*, 2010, pp. 1–4.

[24] IETF, *Key words for use in RFCs to Indicate Requirement Levels*, March 1997, rFC 2119.

[25] W3C, *Web Services Description Language (WSDL) 1.1*, March 2001.

[26] ——, *Web Services Addressing 1.0 - Core*, May 2006.

[27] C. Guidi, I. Lanese, F. Montesi, and G. Zavattaro, "On the Interplay Between Fault Handling and Request-Response Service Interactions," in *ACSD*, Xi'an, China, June 2008, pp. 190–198.

[28] J. Mendling, K. Lassen, and U. Zdun, "On the Transformation of Control Flow between Block-Oriented and Graph-Oriented Process Modeling Languages," *IJBPIM*, vol. 3, no. 2, pp. 96–108, 2008.

[29] OMG, *Business Process Model and Notation (BPMN) Version 2.0*, January 2011.