Towards a Robustness Evaluation Framework for BPEL Engines

*Simon Harrer and Guido Wirtz
Distributed Systems Group
Faculty of IS and Applied CS
University of Bamberg
An der Weberei 5, 96047
Bamberg, Germany
{*simon.harrer,guido.wirtz*}*@uni-bamberg.de*

*Faris Nizamic and Alexander Lazovik
Distributed Systems Group
Johann Bernoulli Institute
University of Groningen
Nijenborgh 9, 9747 AG
Groningen, The Netherlands
{*f.nizamic,a.lazovik*}*@rug.nl*

***Abstract*—The selection of the best fitting process engine for a specific project requires the evaluation of engines according to various requirements. We focus on the non-functional requirement robustness, which is critical in production environments but hard to determine. Thus, we propose an evaluation framework to reveal important robustness criteria of process engines. In this work, we focus on message robustness, i.e., the ability to handle the receipt of invalid messages appropriately. In a case study comprising five open source BPEL engines, we determine message robustness by injecting faults into robustly designed processes as a reply to a previously sent request from an external virtual service and assert their behavior. The results show that the degree of message robustness significantly differs, hence, robustly designed processes do not necessarily lead to robust runtime behavior, the selected engines still play a major role.**

***Keywords*-virtual services, BPEL, process engines, testing, robustness**

I. Introduction

With the rise of business processes throughout industry and academia, process languages and their execution environments, process engines, have become ubiquitous within IT landscapes. According to market research [1], this has lead to a $2.8 billion business process management (BPM) industry in 2013, with an estimate of up to $8.3 billion in 2019. Especially executable business processes are on the rise [2]. There are many executable process languages, e.g., the Web Services Business Process Language 2.0 (WS-BPEL, or BPEL for short) [3] and the Business Process Model and Notation 2.0 (BPMN) [4]. And even more proprietary as well as open source process engines that can execute processes defined in these process languages. As a result, there is a need to compare and choose the process engine which is the best fit for a specific project. While there are many comparison criteria, e.g., conformance, performance, security or installability, we focus on robustness as an essential non-functional requirement in a multitude of production environments, which is seen as a very important property of process engines [5]. Robustness is the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions [6]. Especially in production environment, this property is very important as not robust systems may crash and fail, causing costs.

The comparison of software in general is a topic that easily turns into discussion that is driven by personal taste and preferences rather than by a comparison of facts. Despite the wide academic interest in such comparisons, little work on software comparison can be found, e.g., the comparison of operating systems [7]. We aim to contribute to software comparison in general by focusing on the area of service-oriented computing, more precisely middleware for service orchestration, and evaluate robustness as one of essential non-functional requirements.

Because robustness itself is a wide term, we have created a robustness framework consisting of six robustness criteria that are relevant for BPEL engines. As part of a case study, we propose a method to evaluate the criterion message robustness which is applied onto five open source BPEL engines in an experiment. Our results of this experiment reveal that there are major issues in message robustness in these engines, while some of them can be countered by countermeasures in the processes themselves.

This paper is organized as follows. First, the related work regarding benchmarking process engines, existing robustness testing techniques and virtual services is given in Section II. Next, we outline the robustness framework for BPEL engines in Section III. Subsequently, we present a case study on evaluating message robustness of open source BPEL engines in Section IV. Finally, we conclude this paper in Section V, including future work.

II. Related Work

Related work comprises benchmarking BPEL engines, robustness evaluations and testing, and virtual services.

The benchmarking of BPEL engines has been focused on evaluating the conformance in terms of feature support [8], expressiveness in terms of workflow pattern support [9], installability in terms of effort of use [10] and performance [11],

*The first two authors have contributed equally to this paper, and should therefore be both considered as first authors.

[12] whereas robustness is only taken into account by [13] despite its importance [5, p. 12]. Kopp et al. [13] focus on fault propagation and evaluate it manually for a single BPEL engine (Apache ODE) according to the implementation of the layers TCP, HTTP, SOAP and WS-Adressing by analyzing its source code. In contrast, we provide a framework for evaluating the robustness of process engines automatically and reproducibly. However, we reuse parts of the fault propagation layers and fault categorizations from Kopp et al. [13] for our framework and case study.

Robustness considerations mostly concentrate on how to create a robust web service using process languages [14]–[20]. Dobson [14] proposes four BPEL-based fault tolerance patterns which found themselves on a variety of existing BPEL activities, including the `faultHandlers` for forward error recovery with their `catch` and `catchAll` activities and the `compensationHandler` for backward error recovery. In our experiment, we only use the forward error recovery and leave backward error recovery for future work. Likewise, these fault handlers in BPEL are used in [17] as part of a framework in which specified fault handling mechanisms can be automatically implemented in BPEL. In [16], [20], these fault handling activities of BPEL are used to build fault-tolerant proxies within BPEL, i.e., monitored and fault-tolerant invocations.

In [21], a very similar approach is presented which evaluates the robustness of a single BPEL process on an unspecified engine with . In contrast, we provide a robustness framework encompassing not only message robustness but also five other criteria. Moreover, we evaluate the robustness of five engines with two BPEL processes systematically, hence, the focus is different as well.

Furthermore, there are multiple papers describing robustness testing techniques, i.e., how to inject faults into the system under test and compare the observed behavior with the expected one [22]. For this work, we are also creating malicious SOAP messages, but instead of aiming to breach any security measurements we are solely interested in the type of observable reaction upon receiving faulty SOAP messages, being either the correct response, timeouts and crashes or ignoring the error. This idea of using faulty messages to detect whether an existing bug is absent in a system under test is called fault-based testing and was developed by Morell [23]. Among other approaches, it can be applied to validate messages and their specifications as well, e.g., the fault-based testing of XML schema definitions [24]–[26], making it suitable for our approach. Such messages are mostly created using methods from either fuzz testing or mutation testing. Miller et al. [27] introduced fuzz testing as a special form of random black box testing. They have fed the system under test with automatically generated inputs and only verified whether the system under test crashes which would mark the failure of the test while any other behavior marks the passing of the test. We deviate from this technique as we do not generate the inputs or messages randomly. Moreover, while a crash or timeout is a failure in our method as well, we focus on whether the fault handling mechanisms of the process engine still work as expected. In contrast to fuzz testing, mutation testing does not generate the inputs randomly but applies mutation operators on source code introducing new bugs, i.e., mutants, which have to be detected, i.e., killed, by the test suite [28]–[30]. Nevertheless, instead of mutating the source code and running the test suite, the ideas of mutation testing itself can be applied to input data as well by mutating the data and feeding the data into the system under test which must detect and reject the faulty input. For example, Xu et al. [31] as well as Lee et al. [32] generate test cases in a form of XML messages by applying mutation operators onto XSDs and DTDs and generating pertubed XML messages from their mutated XSDs and DTDs respectively. Franzotte et al. [33] apply mutation testing to XML schemas.

Finally, we also list the work related to the Service Virtualization, i.e., the practice of capturing and simulating the behavior, data, and performance characteristics of dependent systems, and deploying a Virtual Service that represents the dependent system without any constraints [34]. This standardization leads to lower creation costs and a higher portability of virtual services, which in turn makes the simulation of external unavailable web services feasible even in complex and dependent IT landscapes. Such a simulation is necessary to reduce risks during testing and to avoid testing in production environments [35]. Ideas on how simulation of dependent web services behavior can ease the software testing process are described in [36]. In the case study of this work, we use the tool *Parasoft Virtualize* [37] to simulate a faulty external partner service in a standardized manner.

III. Robustness Framework for BPEL Engines

The robustness framework consists of six criteria that we consider being the most important aspects of robustness of BPEL engines. We derived these aspects from the typical execution architecture of BPEL engines as shown in Fig. 1. The gray box denotes the BPEL engine which allocates resources, i.e., CPU, memory and disk, to execute instances of a deployed process. The instances can exchange messages with external services as well as their callers. Moreover, they use resources of the engine to be executed, and can be passivated to a persistent storage to save the state of an instance so that it can be recovered later on. We abstracted from engine components, e.g., a message router and dispatcher, as this is not relevant for our robustness evaluation. Each robustness criterion is linked to an edge in the image and displayed in bold and upper case. While BPEL has complex parallel activities and a complex message correlation mechanism, the evaluation of these is part of a functional evaluation. Because of this, we did not include it in our robustness framework. The criteria are outlined as follows.

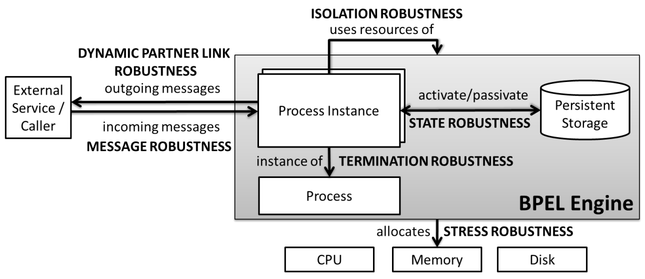

Figure 1. Robustness Criteria Derived Using BPEL Engine Architecture

Message robustness refers to the ability to handle incoming faulty messages appropriately in process instances. Appropriately in this case means that the process instance can react upon this error, inappropriately would be when the process instance crashes, times out or the fault is undetected. Messages can either be received normally from the front door when a system sends a request to the process instance, or from the backdoor as a response from an external partner service. As the incoming messages are SOAP messages transmitted over HTTP, faults can occur on three levels, namely, HTTP, SOAP and XML payload of the SOAP message, e.g., XML and SOAP is not well-formed XML code or not valid to their corresponding XSD, or the wrong status code or MIME/TYPE is transmitted as part of the HTTP header. As the messages are transported over the network, we include transport level issues as well as part of this criterion, e.g., TCP timeouts or DNS problems.

Dynamic partner link robustness refers to the ability to handle bad partner link endpoints appropriately [38], [39]. In BPEL, partner links can be set dynamically at runtime for external services, hence, they can be set to a bad value at runtime. The engine is considered robust when it provides the process instance with the ability to react upon the usage of bad partner link endpoints.

Stress robustness refers to the ability of the engine to react upon a major increase in workload with their allocated resources, namely, CPU time and memory/disk space. An engine is considered robust in an absolute way when an arbitrary large workload cannot crash the engine or render it unresponsive. An engine is considered more robust relative to another engine when it can handle more stress before crashing or becoming unresponsive.

State robustness refers to ability to handle crashes of the process engine in a way so that the administrator can recover the current instance data. An engine is considered robust when an arbitrary large workload cannot set the engine in a broken and inconsistent state. It may slow down its performance, or even crash it, as long as the state of the engine is still consistent and recoverable. For example, when the virtual machine stops working with an out of memory error, the engine should make sure that the execution state of the processes has been safely passivated to disk which can be stored to recover from this error.

Termination robustness refers to the ability to detect issues that avoid the termination of process instances. This comprises the detection of deadlocks, e.g., unreachable activities [40], or livelocks, e.g., an infinite loop, at either compile time or runtime. The more of these situations an engine is able to detect and react upon, the more robust it can be considered. At compile time, an appropriate reaction is to inform the user about the issue and optionally to reject the faulty process. At runtime, an appropriate reaction is to inform the user about the issue and optionally halting in case of a livelock or terminating the process instance in case of a deadlock automatically.

Isolation robustness refers to the ability to isolate the execution of a single process instance to ensure that such an instance cannot affect the whole system. For example, when a process instance has been caught up in an infinite loop it should not be able to crash the engine or render it unresponsive. To be considered robust, the engine has to prevent situations like this.

IV. Case Study: Evaluating Message Robustness

In this section, we present a case study on how to evaluate message robustness of open source BPEL engines. First, we propose a method to evaluate the message robustness of BPEL engines in Section IV-A. In Section IV-B, we describe the experiment to validate our proposed method and present its results in Section IV-C. Finally, we discuss the findings and threats to validity in Section IV-D.

A. Method

To evaluate message robustness properties of a BPEL engine, we propose to observe the runtime behavior of instances of robust processes that interact with faulty partner services and determine whether the process instances are able to react upon the faulty response of the partner service. We define this kind of message robustness as *backdoor message robustness* as the faults are injected as response of an external third party service. This is in contrast to *frontdoor message robustness* in which the faulty messages are sent directly as requests to an existing instance or creating a new one. Robust processes use fault handling and validation constructs available in the process language which would achieve fault tolerance when executed on a robust engine. Hence, we propose to create a process for each of those constructs to evaluate the actual message robustness capabilities of each of those constructs of an engine. In addition, the partner services have to be configured to respond with faults that are suitable to reveal message robustness issues. We propose to create false responses systematically for each of the layers. Moreover, the false responses are created by applying one mutation to the correct response to test each fault in isolation of each other. A test case is the combination of using a robust process with a specific fault handling construct that needs to handle a specific faulty response on a specific engine. When a fault is handled by at least one robust process, the engine is considered robust regarding this faulty response. Upon this data, we can derive message robustness properties per layer.

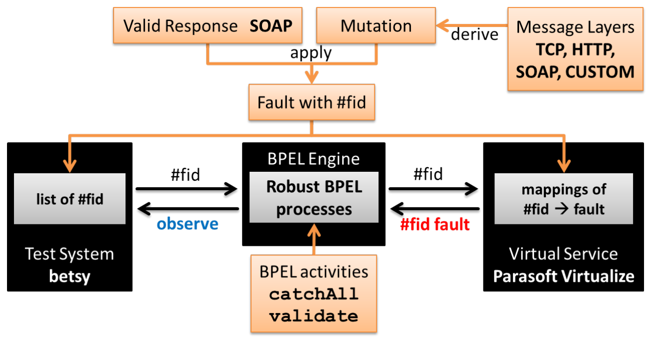

Figure 2. Big Picture

1) *Big Picture:* The big picture of the test architecture of this method is shown in Fig. 2 and comprises three components (black boxes) and their configuration (gray boxes): the *test system* betsy which requires the *list of all fault ids*, *BPEL engine* under test onto which the *robust processes* need to be deployed and the *virtual service* build with Parasoft Virtualize which has to be parametrized with the *request-response mappings* or pairs of the fault id and the corresponding fault. The orange arrows denote how the robust processes and how the faults and their fault ids (#fids) are created. The robust processes are created by using activities from the process language that enable fault handling and error recovery. In contrast, the creation of the faults requires multiple steps. First, for each specification of each message layer, mutations are derived on how to change a valid message to an invalid one. Second, these mutations are applied to a normative valid response. Third, each fault is assigned an identifier, a fault id, that corresponds to the fault, making up a request-response pair. The black arrows mark the message flow per test execution between the components. A test case is initiated by the test system by sending a fault id to a deployed robust process and observes the response which is used to determine the message robustness. The virtual service, however, responds to requests of the process instances by returning the fault corresponding to the sent fault id.

2) *Robust Processes:* We use two robustly designed processes, one being an extension of the other, based upon the process stub from [8, p. 4]. They are shown in Listing 1 and presented in pseudo code that uses indentation to denote hierarchy, marks BPEL activities as bold and dollar prefixes variables. Line 6 and 15 represents the `receive-reply` pair to observe the behavior of the `scope` encompassing the lines 7 to 14 in which the faulty partner service is called synchronously using the `invoke` activity and then sets the `$result` variable to `NO_FAULT`. To the `scope` itself, a `catchAll` fault handler is attached which sets the `$result` variable to `FAULT` in case it catches any fault. The `catchAll` activity is specified "to catch any fault not caught by a more specific fault handler" [3, p. 128], hence, it is the central point of creating a robust and fault tolerant process as it allows reacting upon an error. By checking the contents of the `$result` variable after process completion, we can observe whether the `catchAll` activity is executed at runtime or not, and consequently determine whether the engine was able to react upon the simulated error or not. The second robust process (RP#2) extends the first robust process (RP#1) by validating the received response against its XSD definition with the `validate` activity in line 13. This is an additional robustness instrument to ensure that the incoming message is validated against its XSD definition, independent of the engine which may or may not validate the incoming message against its XSD definition.

Listing 1. Robust BPEL Process #1 and RP#2

```
 1 process
 2   imports
 3   partnerLinks
 4   variables $id, $result, $response
 5   sequence
 6     receive $id
 7     scope
 8       faultHandlers
 9         catchAll
10           assign FAULT to $result
11       sequence
12         invoke faulty service synchronously by sending $id
                and wait for $response
13         [only RP#2: validate $response against schema]
14         assign NO_FAULT to $result
15     reply $result
```

The activities of RP#1 are widely used in real world process. According to [2], "more than 70% of [real world] processes contain fault handlers" [2, p. 7] as well as "invoke, sequence, assign and receive occur in more than 93% of processes" [2, p. 7]. The `validate` activity of RP#2 is not used in any of the real world processes stated in [2], hence, indicates that this countermeasure is not widely applied.

While there is the `catch` activity as well to handle a single and specified fault, we cannot apply this activity as in our experiment we want evaluate the ability to catch undefined and unspecified faults, i.e., no predefined SOAP faults. The processes are considered robust as they try use fault handling logic to cope with an erroneous response by themselves without intervention of an administrator.

3) *Test Suite:* A test suite is a collection of test cases that are intended to be used to test a software program to verify that it has some specified set of behaviors. Hence, to verify whether the five BPEL engines under test have a satisfying degree of message robustness, we created 75 test cases which are shown in Table I. The test cases are structured according to the layers in which an error may occur, being either on the lowest level (3 faults), in the HTTP header (40 faults), in the

XML-based SOAP envelope (21 faults) or in the application specific part within the SOAP body (11 faults). Apart from the layer, the tests do have an additional type depending on which specification is used to derive the faults from. The tests have been created manually.

TCP Test Cases: The three test cases on the TCP layer refer to not being able to resolve the DNS entry and having either an unreachable or an non-responding host. These tests are based on the faults defined in [13]. They are simulated by changing the endpoint of the external partner service according to the specific fault, and are therefore the only three test cases that are not on the message level but on the transport level.

HTTP Test Cases: For the HTTP layer, we solely created tests for different status codes of the HTTP header by sending the correct SOAP payload but only changing the status code. As the first digit of the HTTP status code determines its type, we subdivided them accordingly. Only the code 200 is removed as this marks a correct response, or in other words, these test cases are mutations of the valid 200 return code.

SOAP and APPDATA Test Cases: The tests from both the SOAP and the APPDATA layer are encoded as mutations of the valid SOAP response. The XML mutations, which refer to the well-formedness criteria of XML, are solely at the SOAP layer while we subdivided the XSD mutations, which refer to the correctness against their XSD schema, into the part referring to SOAP XSD schema and the XSD schema of the application specific code. We extracted the XML mutations out of the XML specification and grouped our tests by bad names (4), unescaped symbols (5), unclosed entities (4), structural errors (1) and root element issues (3). For the XSD mutations, we permutated the operator action (add, remove and change) and the operator target (namespace, namespace prefix, element, content, attribute and text), removed the meaningless combinations and created test cases for the meaningful ones. The issue is, however, that not every meaningful combination can be applied to both the SOAP and the APPDATA variant as not every mutation makes sense. Therefore, the SOAP XSD test cases are fewer than the ones for the APPDATA layer.

B. Experiment

To validate our proposed method from Section IV-A, we adopt the generic architecture of the method as shown in Fig. 2 and applied it onto testing BPEL engines. We use the *BPEL engine test system* (betsy) as the test system, five different BPEL engines (see Section IV-B2), and *Parasoft Virtualize* as our virtual service (see Section IV-B3).

For the experiment itself, we have set up a single machine locally on which both betsy and the BPEL engines have been running while *Virtualize* has been provided via an internet-based virtual machine of an Infrastructure as a Service (IaaS) provider.

1) BPEL Engine Test System (betsy): For the test execution, we adopted the open source tool *betsy* [41] for our purposes[1]. Because of this, we can run the whole experiment automatically as betsy installs, starts and stops the BPEL engines as well as deploys the BPEL processes, tests their behavior via soapUI[2] and creates detailed reports. In addition, we get test isolation for free because betsy provides a fresh instance of the engine under test for each test case by means of virtualization [42].

2) BPEL Engines: In this experiment, we evaluate the message robustness of five open source BPEL engines: *Apache ODE* v1.3.5, *bpel-g* v5.3, *Orchestra* v4.9, *Petals ESB* v4.1 and *OpenESB* v2.2. All of these engines are implemented in Java and run on either a light-weight servlet container, e.g., an Apache Tomcat, or an enterprise service bus (ESB), e.g., the Glassfish Application Server. These engines are suitable to evaluate our method as they use various Web Service stacks as well as ship their very own BPEL engine implementation.

3) Virtual Service: To inject faulty messages into our BPEL processes, we created a virtual service with Parasoft *Virtualize* v9.6 [37], a proprietary software product that can create, deploy, and manage simulated test environments mainly for purposes of software development and testing. To set up the virtual service, we created a *Virtual Asset*, i.e., an invocable and simulated web service, that for predefined request parameters (*fault id* or *#fid* for short) returns predefined faulty responses (*faults*). All different types of faults that we simulate are shown as part of Table I. As the fault ids are only technically, we omitted them from the tables.

C. Results

The results of our experiment are shown in Table I. Each row represents either a single test case or multiple test cases with the same results. The rows themselves are grouped hierarchically depending on the layer and the type of the mutation. Cells within each row mark the result of this test case of a specific engine, which can either be + (robust; process can react upon the error), T (timeout), R (regular response) or R_+ (regular response for RP#1, robust for RP#2), which is observed from the test system.

The baseline test case that uses the valid response as well as a valid SOAP fault execute correctly on all five engines under test for the RP#1. But as not every engine supports the `validate` activity according to [8], this does not hold for the robust BPEL process #2. Therefore, we present the results of the RP#1 while the results regarding RP#2 are shown as subscripts in case the `validate` activity was able to achieve a more robust test result. For this experiment, only OpenESB has any gains when using the `validate` activity.

[1]Source code: https://github.com/uniba-dsg/betsy/tree/ase2014
[2]http://www.soapui.org/

Table I
ROBUSTNESS RESULTS

			bpel-g	Apache ODE	OpenESB	Orchestra	Petals ESB
APPDATA	XSD	remove element	+	T	+	T	T
		remove content	+	R	R_+	R	R
		change int to string; int to localized double	+	R	R_+	R	R
		change int to double	+	R	R	R	R
		no or wrong namespace	+	T	+	R	R
		unbound namespace prefix	+	T	+	+	T
		add element	R	R	R	T	R
		add attribute	+	R	R_+	R	R
		add text between elems	+	R	+	R	R
SOAP	XML	no root element; text instead of root element	+	T	+	+	T
		two root elements	+	R	+	R	R
		elements overlap	+	T	+	+	T
		unclosed element	+	R	+	R	R
		unclosed attribute, comment, or CDATA	+	T	+	+	T
		unescaped $<$ or &	+	R	+	+	T
		unescaped $>$, ', or "	R	R	+	R	R
		name starts with XML	+	T	+	R	T
		name contains space; starts with num or dash	+	T	+	+	T
	XSD	remove element	+	T	+	T	T
		no or wrong namespace; unbound namespace prefix	+	T	+	+	T
HTTP	Status Code	100	T	T	+	T	T
		101	T	T	+	R	T
		201; 203; 206	R	T	R	R	T
		202	R	T	+	R	T
		204	+	T	+	+	T
		205	T	T	T	T	T
		300	+	T	R	R	T
		301…303; 305…307	+	T	+	R	T
		304	+	T	+	+	T
		400	+	R	+	T	R
		401…417	+	T	+	T	T
		500	R	R	R	R	R
		501…505	R	T	R	T	T
TCP	DNS	unresolvable	+	T	+	T	T
	HOST	unreachable; resp. timeout	T	T	T	T	T

1) TCP layer: The observed engine behavior for this layer is to time out, except for bpel-g and OpenESB which can handle unresolvable DNS issues.

2) HTTP layer: Regarding the HTTP layer, we test the different status codes of the HTTP header. There are three ways an engine handles faulty status codes, it correctly raises a catchable exception (+), it times out (T), or it ignores the header data and carries on using the HTTP payload (R). The status code tests are grouped using the first digit of the status code, namely, 1xx, 2xx, 3xx, 4xx and 5xx. For 1xx status codes, only OpenESB reacts as expected, while the other engines time out, except for Orchestra which ignores the status code 101. For 2xx status codes, all engines fail to respond correctly, except bpel-g for status code 204, OpenESB for status code 202 and 204, and Orchestra for 204 status code. For 3xx status codes, bpel-g and OpenESB react as expected except for a 300 which is ignored in OpenESB. Apache ODE and Petals ESB time out, and Orchestra always ignores the status code except for a 304 which it handles correctly. For 4xx codes, bpel-g and OpenESB detect all the faults whereas Apache ODE, Orchestra and Petals ESB always time out except for status code 400 which is ignored by Apache ODE and Petals ESB. No engine handles 5xx status codes correctly. bpel-g and OpenESB ignore the status code while the other three engines time out. An exception is status code 500 which all engines ignore. The engines clearly violate the SOAP specification, as it states that the return code 500 must only be used for SOAP faults and not for normal SOAP messages [43, section 6.2]. Overall, the behavior of both Apache ODE and Petals ESB is identical for all status codes, hinting to the fact that they used the same library for handling such codes. Orchestra has the same behavior for the 5xx and most of the 4xx tests. In comparison, bpel-g and OpenESB share quite the same behavior as well, only the four status codes 100, 101, 202 and 300 out of the 41 (approx. 10 %) are handled differently, hinting at the fact that they used the same library slightly differently.

3) SOAP layer: For SOAP XML content, the results vary substantially. Only OpenESB handles all test cases correctly, while both Apache ODE and Petals ESB fail in all cases. It is interesting to note that both bpel-g and Orchestra may either succeed or return a regular response for each test, but never time out. In fact, they almost have the same pattern, except for three cases (two root elements, unclosed element and name starts with XML), in which Orchestra returns regular responses instead. Apache ODE and Petals ESB either respond with a timeout or regular response. Interestingly, both Apache ODE and Petals ESB time out for all the cases in which there is a malformed name in the XML file. Overall, there are no tests which are handled equally on all engines, but there are multiple test cases in which the fault is ignored, especially within the test case group for unescaped symbols, marking an area which seems to be more problematic than other SOAP XML test cases. In addition, the other remaining tests that return a regular response may have automated corrections built into the XML parsing libraries, e.g., removing a second root element while parsing. For SOAP XSD content, bpel-g and OpenESB respond correctly for all XSD mutations, Orchestra times out only for one test case (remove element), while Apache ODE and Petals ESB time out for all the cases. To sum up, these test cases are handled either correctly are with a time out, no regular response is given.

4) APPDATA layer: bpel-g succeeds in every test except for the *add element* test case, in which a regular response is returned. Apache ODE and Petals ESB did not return any correct response, they either time out or return a regular response. For the test cases *change int to string*, *double* or *localized double*, and add of element, attribute or text between elements, both engines only return regular responses. Orchestra responds with a regular response for all the cases except for addition and removal of an element, in which case it times out. The only test Orchestra is able to detect is *unbound namespace prefix*. OpenESB is special in this context, as five of the cases of the test run with process #1 return correct responses and the other six return the wrong regular responses. However, when rerunning these failed test cases with the second process, four test cases (remove content, change int to string, change int to localized double and add attribute) are now succeeding and marked with (R_+), i.e., the `validate` activity had an impact for these tests.

D. Discussion

In this section, we provide an overall BPEL engine comparison in terms of message robustness (see Section IV-D1) followed by a discussion of possible threats to validity of our approach (see Section IV-D2).

1) Engine Comparison: The overall and aggregated results are shown in Table II. For each of the four layers, the percentage of the number of successful tests subdivided by the total number of tests per layer is given for the robust BPEL process #1 of the five BPEL engines. As the robust BPEL process #2 could only improve the message robustness results of OpenESB, these percentages are shown in the last column marked with an asterisk. The numbers show clearly that both Petals ESB and Apache ODE do not handle any of our test cases robustly while the remaining three handle the errors quite differently ranging from 5% up to 100%. Orchestra always performs worse than bpel-g and OpenESB in any of the layers. In contrast, OpenESB always performs best when looking at the TCP (33%), HTTP (73%) and the SOAP (100%) layer, but cannot compete with bpel-g in terms of APPDATA message robustness (45% vs. 91%). However, when using RP#2, OpenESB is able to react upon 82% of the APPDATA test cases, and closes up to bpel-g in this layer. Consequently, OpenESB and bpel-g seem to be the most robust BPEL engines of the five we evaluated, while OpenESB is more robust than bpel-g according to our test results. Nevertheless, we do not create an overall ranking as the number of test cases vary greatly and computing an overall percentage would yield a meaningless value.

2) Threats to Validity: The threats to validity of our experiment can be subdivided into issues regarding the test cases, the robust processes and engines under test. The execution of the experiment itself has been fully automated while the creation of the test cases was done manually. Especially the implementation of the SOAP and APPDATA test cases required manual modeling of errors. To reduce as many flaws in these tests as possible, we applied a peer-review of the tests itself by the authors and an XML tool `xmllint` for both well-formedness and schema compliance checks to ensure that each test case encompasses exactly its single introduced fault. The robustly designed BPEL processes only make use of the forward error recovery activities. Still, our experiment shows that our method is on the right track, but for a more extensive study, we aim to evaluate the effectiveness of the backward error recovery activities as well, especially in the case when forward error recovery fails to work, i.e., building upon the results of the experiment in this work. In our experiment, we evaluated five open source BPEL engines. While we used the latest version of Orchestra and bpel-g, we only used the next-to-last version of Apache ODE, OpenESB and Petals ESB. As this experiment only proofs the feasibility and usefulness of our robustness evaluation method, it is not necessary to conduct the study with the most recent version. Additionally, the engines under test in the experiment are all open source, not taking proprietary engines into account which are often considered to be more mature and robust. As these open source ones are used in production as well, they still make up as a relevant set of targets in our experiment, leaving the comparison of open source and proprietary ones for future work.

Table II
ROBUSTNESS DEGREES PER LAYER AND ENGINE

Engine / Layer	bpel-g	Apache ODE	OpenESB	Orchestra	Petals ESB	OpenESB*
APPDATA	91%	0%	45%	9%	0%	82%
SOAP	86%	0%	100%	67%	0%	100%
HTTP	68%	0%	73%	5%	0%	73%
TCP	33%	0%	33%	0%	0%	33%

V. CONCLUSION AND FUTURE WORK

In this work, we presented a robustness framework defining the relevant robustness criteria for BPEL engines. As part of a case study, we presented a method to evaluate backdoor message robustness of BPEL engines via an experiment with five open source engines. The results reveal that the engines do have issues when receiving erroneous messages, leading to a lot of manual intervention for their operators. On OpenESB, few issues can be mitigated by leveraging the `validate` construct of BPEL in the process itself after receiving the messages.

Future work comprises the creation of an approach for all robustness criteria, taking proprietary engines into account, and applying this robustness approach onto BPMN [4] engines as well.

ACKNOWLEDGEMENTS

The authors would like to thank to Dr. Rix Groenboom from Parasoft Netherlands for fruitful discussions on the

topic and also for providing multiple licenses for Parasoft products.

References

[1] Wintergreen Research, "Business Process Management (BPM) Cloud, Mobile, and Patterns: Market Shares, Strategies, and Forecasts, Worldwide, 2013 to 2019," Market Research, 2013.

[2] M. Hertis and M. B. Juric, "An Empirical Analysis of Business Process Execution Language Usage," *IEEE Transactions on Software Engineering*, vol. 40, no. 8, pp. 738–757, Mai 2014.

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

[4] OMG, *Business Process Model and Notation*, January 2011, v2.0.

[5] F. Leymann, "BPEL vs. BPMN 2.0: Should You Care?" in *Business Process Modeling Notation*, ser. Lecture Notes in Business Information Processing, J. Mendling, M. Weidlich, and M. Weske, Eds. Springer Berlin Heidelberg, 2010, vol. 67, pp. 8–13. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-16298-5_2

[6] IEEE, "IEEE Standard Glossary of Software Engineering Terminology," *IEEE Std 610.12-1990*, pp. 1–84, December 1990.

[7] D. Spinellis, "Quality wars: Open source versus proprietary software," in *Making Software: What Really Works, and Why We Believe It*, A. Oram and G. Wilson, Eds. Sebastopol, CA: O'Reilly and Associates, 2010, ch. 15, pp. 259–293. [Online]. Available: http://www.spinellis.gr/pubs/inbook/2010-MakingSW-QualityWars/html/Spi10k.html

[8] S. Harrer, J. Lenhard, and G. Wirtz, "BPEL Conformance in Open Source Engines," in *Proceedings of the 5th IEEE International Conference on Service-Oriented Computing and Applications (SOCA'12), Taipei, Taiwan*. IEEE, 17–19 December 2012, pp. 1–8.

[9] ——, "Open Source versus Proprietary Software in Service-Orientation: The Case of BPEL Engines," in *ICSOC*, ser. Lecture Notes in Computer Science, vol. 8274. Berlin, Germany: Springer Berlin Heidelberg, 2013, pp. 99–113.

[10] J. Lenhard, S. Harrer, and G. Wirtz, "Measuring the Installability of Service Orchestrations Using the SQuaRE Method," in *Proceedings of the 6th IEEE International Conference on Service-Oriented Computing and Applications (SOCA'13)*. Kauai, Hawaii, USA: IEEE, December 16-18 2013.

[11] C. Röck, S. Harrer, and G. Wirtz, "Performance Benchmarking of BPEL Engines: A Comparison Framework, Status Quo Evaluation and Challenges," in *26th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, Vancouver, Canada, July 2014, pp. 31–34.

[12] D. Bianculli, W. Binder, and M. L. Drago, "SOABench: Performance Evaluation of Service-oriented Middleware Made Easy," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 301–302. [Online]. Available: http://doi.acm.org/10.1145/1810295.1810361

[13] O. Kopp, F. Leymann, and D. Wutke, "Fault Handling in the Web Service Stack," in *Service-Oriented Computing*, ser. Lecture Notes in Computer Science, P. Maglio, M. Weske, J. Yang, and M. Fantinato, Eds. Springer Berlin Heidelberg, 2010, vol. 6470, pp. 303–317. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-17358-5_21

[14] G. Dobson, "Using WS-BPEL to Implement Software Fault Tolerance for Web Services," in *Software Engineering and Advanced Applications, 2006. SEAA'06. 32nd EUROMICRO Conference on*. IEEE, 2006, pp. 126–133.

[15] S. Samaranayake, R. Jiménez-Peris, and M. Patiño-Martínez, "Fault-Tolerant Business Processes," *Jornadas de Ciencia e Ingeniería de Servicios (JCIS)*, 2011.

[16] O. Ezenwoye and S. M. Sadjadi, "Robustbpel2: Transparent autonomization in business processes through dynamic proxies," in *Autonomous Decentralized Systems, 2007. ISADS'07. Eighth International Symposium on*. IEEE, 2007, pp. 17–24.

[17] A. Liu, Q. Li, L. Huang, and M. Xiao, "Facts: A framework for fault-tolerant composition of transactional web services," *Services Computing, IEEE Transactions on*, vol. 3, no. 1, pp. 46–59, Jan 2010.

[18] P. P. W. Chan, M. R. Lyu, and M. Malek, "Making Services Fault Tolerant," in *Service Availability*. Springer, 2006, pp. 43–61.

[19] G. Alonso, A. El Abbadi, C. Mohan, C. Hagen, and D. Agrawal, "Enhancing the fault tolerance of workflow management systems," *IEEE Concurrency*, vol. 8, no. 3, pp. 74–81, 2000.

[20] O. Ezenwoye and S. M. Sadjadi, "Enabling Robustness in Existing BPEL Processes," in *ICEIS*, 2006, pp. 95–102.

[21] S. Ilieva, D. Manova, I. Manova, C. Bartolini, A. Bertolino, and F. Lonetti, "An Automated Approach to Robustness Testing of BPEL Orchestrations," *2014 IEEE 8th International Symposium on Service Oriented System Engineering*, vol. 0, pp. 193–203, 2011.

[22] A. Mukherjee and D. Siewiorek, "Measuring Software Dependability by Robustness Benchmarking," *Software Engineering, IEEE Transactions on*, vol. 23, no. 6, pp. 366–378, Jun 1997.

[23] L. J. Morell, "A Theory of Fault-Based Testing," *IEEE Trans. Softw. Eng.*, vol. 16, no. 8, pp. 844–857, Aug. 1990.

[24] M. C. F. P. Emer, I. F. Nazar, S. R. Vergilio, and M. Jino, "Fault-Based Test of XML Schemas," *Computing & Informatics*, vol. 30, no. 3, 2011.

[25] M. Emer, S. Vergilio, and M. Jino, "A Testing Approach for XML Schemas," in *COMPSAC*, 2005.

[26] J. B. Li and J. Miller, "Testing the Semantics of W3C XML Schema," in *COMPSAC*, 2005.

[27] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990. [Online]. Available: http://doi.acm.org/10.1145/96267.96279

[28] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[29] J. Offutt, "A Practical System for Mutation Testing: Help for the Common Programmer," in *ITC*, 1994.

[30] J. Offutt and W. Xu, "Generating Test Cases for Web Services Using Data Perturbation," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 5, pp. 1–10, Sep. 2004.

[31] W. Xu, J. Offutt, and J. Luo, "Testing Web Services by XML Perturbation," in *ISSRE*, 2005.

[32] S. C. Lee and J. Offutt, "Generating Test Cases for XML-based Web Component Interactions Using Mutation Analysis," in *ISSRE*, 2001.

[33] L. Franzotte and S. R. Vergilio, "Applying Mutation Testing in XML Schemas," in *SEKE*, 2006, pp. 511–516.

[34] S. Strauch, V. Andrikopoulos, S. G. Saéz, and F. Leymann, "Implementation and evaluation of a multi-tenant open-source esb," in *Proceedings of the European Conference on Service-Oriented and Cloud Computing, ESOCC'13, 2013*, ser. Lecture Notes in Computer Science (LNCS), vol. 8135. Springer Berlin Heidelberg, 2013, pp. 79–93.

[35] F. Nizamic, R. Groenboom, and A. Lazovik, "Testing for highly distributed service-oriented systems using virtual

environments," in *Postproceedings of 17th Dutch Testing Day*, 2011. [Online]. Available: http://eprints.eemcs.utwente.nl/21523/

[36] F. Nizamic, "Testing of Distributed Service-Oriented Systems," in *ICSOC*, 2013.

[37] Parasoft, *Parasoft Service Virtualization*, http://www.parasoft.com/service-virtualization.

[38] W. T. Tsai, W. Song, R. Paul, Z. Cao, and H. Huang, "Services-Oriented Dynamic Reconfiguration Framework for Dependable Distributed Computing," in *COMPSAC*, 2004, pp. 554–559.

[39] W. Hummer, O. Raz, O. Shehory, P. Leitner, and S. Dustdar, "Testing of Data-Centric and Event-Based Dynamic Service Compositions," *Software Testing, Verification and Reliability*, vol. 23, no. 6, pp. 465–497, 2013.

[40] C. Ouyang, E. Verbeek, W. M. Van Der Aalst, S. Breutel, M. Dumas, and A. H. Ter Hofstede, "Formal Semantics and Analysis of Control Flow in WS-BPEL," *Science of Computer Programming*, vol. 67, no. 2, pp. 162–198, 2007.

[41] S. Harrer and J. Lenhard, "Betsy–A BPEL Engine Test System," Otto-Friedrich Universität Bamberg, Tech. Rep. 90, July 2012.

[42] S. Harrer, C. Röck, and G. Wirtz, "Automated and Isolated Tests for Complex Middleware Products: The Case of BPEL Engines," in *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on*, Cleveland, Ohio, USA, April 2014, pp. 390 – 398, testing Tools Track.

[43] W3C, *Simple Object Access Protocol (SOAP) 1.1*, 2000.