BPEL Conformance in Open Source Engines: The Case of Static Analysis

Simon Harrer, Christian Preißinger and Guido Wirtz
Distributed Systems Group
University of Bamberg
Bamberg, Germany
{*simon.harrer,guido.wirtz*}*@uni-bamberg.de*
{*christian.preissinger*}*@morsepost.de*

***Abstract*—In 2007, OASIS finalized their Business Process Execution Language 2.0 (BPEL) specification which defines an XML-based language for building orchestrations of Web Services. As the validation of BPEL processes against the official BPEL XML schema leaves room for a plethora of static errors, the specification contains 94 static analysis rules to cover all static errors. According to the specification, any violations of these rules are to be checked by a standard conformant engine at deployment time. When a violation is not detected in BPEL processes during deployment, such errors remain unnoticed until runtime, making them expensive to find and fix. In this work, we investigate whether mature BPEL engines that claimed standard conformance implement these static rules. To answer this question, we formalize the static rules and derive test cases based on these formalizations to evaluate the degree of support for static analysis of six open source BPEL engines using the BPEL Engine Test System (*betsy*). In addition, we propose a method to get more accurate static analysis conformance results by taking the feature conformance of engines into account to exclude false positives in contrast to the classic approach. The results reveal that support for static analysis in these engines varies greatly, ranging from nonexistent to full support. Furthermore, our proposed method outperforms the classic one in terms of accuracy.**

***Keywords*-BPEL, engine, conformance testing, static analysis**

I. Introduction

The Business Process Execution Language (BPEL), a standard by OASIS [1], defines a graph and block structured process language (see [2]), and corresponding execution semantics. BPEL is based on XML and relies on XML schema definitions (XSD) [3] which restrict the XML elements and attributes that can be used in a process. Processes have activities and orchestrate message exchanges between services defined in Web Service Definition Language (WSDL) [4] interfaces. Thus, in a Web Service based service-oriented architecture (SOA), BPEL is *the* solution for service orchestration [5], as a BPEL process consumes external web services and exposes itself as one or more web services as well through partner links. The block structure is formed by activities like `<process>` and `<scope>` that encompass other activities and can define fault, compensation, and termination handlers (FCT handlers) which deal with the occurrences of internal and external errors, compensate any previously initiated external operation, or clean up anything before the process is terminated. Event handlers are triggered by timers or incoming messages and define their own enclosing `<scope>`. The first activity of a process is the *start activity* that receives a message. The information from such a message is assigned to `variables` and is used for other data flow activities or message activities. It can also be used in `correlations` that allows multiple message exchanges for a single process instance. Loop blocks are also available, as well as `<flows>` that maintain graph-based control flow structures through links. The BPEL standard defines 94 static analysis rules[1] that restrict these activities. These rules extend the language restrictions of the XSD defined in the BPEL specification. If a process violates a rule, a BPEL engine is assumed to reject it. In case an engine does not reject such a process, the error may only be found at runtime, causing high costs to find and fix the error [6].

The language syntax and semantics of BPEL are topic of extensive research. Related work focuses on models or static analysis itself, while we focus on benchmarking the standard conformance of process engines. *Standard conformance* of an engine can be subdivided into *feature conformance* and *static analysis conformance*. Feature conformance has already been evaluated in [7], and in this case, we benchmark engines to determine their static analysis conformance with fault tests as part of the overall BPEL standard conformance evaluation. In this work, we are only interested in executable processes. Without executable processes and engines, the usefulness of a language like BPEL is limited to modeling purposes only. Hence, correct engine behavior is important for practical application. Therefore, we concentrate on executable processes exclusively and neglect opaque XML nodes as well as abstract processes. The static analysis conformance of an engine can be measured by the result of a fault test deployment. We denote this as the *classical approach* of static analysis evaluation. Based on previous findings in [7] regarding feature conformance we know that support for BPEL features varies greatly between available engines. On the one hand, this indicates a likelihood of a varying support

[1]The static analysis rules are numbered from 1 to 95. In total, there are 94 rules because the rule 49 is missing.

for static analysis by the engines which in the best case reject all invalid processes [1, p. 194]. On the other hand, the rejection of a process is not necessarily caused by an erroneous process, but may be caused by an unsupported feature. To counter this, we propose a *pairwise approach* which uses the pair of a feature and a fault test instead of a sole fault test, taking feature conformance into account. This allows excluding false positives of the fault test based on the result of the corresponding feature test for each test pair.

Hence, we tackle the following research questions:
RQ1: How good is the support for static analysis rules predefined by the standard in open source BPEL engines?
RQ2: Does the pairwise approach outperform the classical approach of measuring static analysis conformance in terms of accuracy?

This work extends and implements the proposal outlined in [8]. This paper itself is structured as follows. In Section II, we outline related work. The approach and the methods used in this paper are given in Section III, followed by a short description of the tests and the open source engines under test in Section IV. In Section V, the results are presented by executing the previously developed test suite against the engines under test, followed by a discussion of these results and their implications in Section VI. The paper concludes and outlines future work in Section VII.

II. Related Work

Static analysis regarding BPEL has been studied extensively in literature. We have gathered the most relevant approaches to this work in Table I. Each approach considered was analyzed by investigating four aspects: the BPEL version, the number of test cases, the amount of static analysis rules covered by the tests, and whether imported WSDL definitions are checked or not. The approaches [9]–[12] focus on BPEL 1.1 whereas [13]–[17] focus on the latest specification BPEL 2.0. Because the rules were initially published in the BPEL 2.0 specification, the first four approaches could not specify any SA conformance tests. Nevertheless, Akehurst [9] and Ouyang et al. [12] provide 16 and 30 valid BPEL 1.1 processes as test cases, respectively. In addition, Ouyang et al. [12] present two incomplete BPEL 1.1 processes detailing an unreachable activity and a conflicting `receive`, the latter would violate the rule #60 of BPEL 2.0 which requires the use of explicit `messageExchanges` in this case. Returning to the approaches using BPEL 2.0, two sources, namely [14] and [15] provide tests. Lohmann [14] provides 56 test cases[2], each of them corresponds to a single static analysis rule. Because of a different focus of [14], the 56 provided tests are not suitable to evaluate the conformance to the static analysis rules of BPEL engines, as they are abstract processes. The BPEL engine Orchestra [15] provides a test set of four valid processes and 92 invalid processes[3]. The invalid ones violate 42 rules in total and have a low quality, as one third violates more than a single rule in every test, and the majority is not valid to the XSD schema of BPEL. As BPEL processes depend on WSDL definitions, it is important to check the validity of imported WSDL files during static analysis of a process. Five approaches [11], [13]–[15], [17] include such a check, but the tests in [14] do not import every WSDL file that is required by the check. The other static analysis approaches [9], [10], [12], [16] validate the BPEL file, only.

Whereas the discussed approaches check the static analysis conformance of BPEL processes, none of them evaluates the static analysis conformance of BPEL engines. Harrer et al. [7], [18] did focus on BPEL engines with their automated testing tool *betsy*, but solely evaluated feature conformance using a large set of valid processes[4]. Thus, standard conformance regarding the static analysis rules of BPEL engines, i.e., static analysis conformance, remains untested [7, p. 7].

Kopp et al. [19] formalize BPEL in a model that includes constraints of 65 static analysis rules, whereas 4 are mentioned but not formalized, 24 rules are declared out of scope in their work, and a single rule (#21) is not mentioned at all. We base our combinations on the model of Kopp et al. [19]. The negation of 50 rule models [19] are identical to our formalization. In contrast, 8 rules are modeled by Kopp et al. [19] that we do not use, and 21 formalizations are insufficient to generate tests, i.e., not existent or not complete. Hence, we had to model them ourselves. Moreover, both Kopp et al. and ourselves excluded 15 rules explicitly because they are either engine-specific or make use of arbitrary XPath expressions[5].

Table I
Approaches Analyzing BPEL Processes or Engines

Approach	BPEL Version	Test Cases Invalid/Valid	SA Rules Covered	Checks WSDL
Akehurst [9]	1.1	- / 16	-	-
Fisteus et al. [10]	1.1	- / -	-	-
Foster et al. [11]	1.1	- / -	-	yes
Gravel et al. [13]	2.0	- / -	-	yes
Lohmann [14]	2.0	56 / -	56	partial
Orchestra [15]	2.0	92 / 4	41	yes
Ouyang et al. [12]	1.1	2 / 30	(1)	-
Yang et al. [16]	2.0	- / -	-	-
Ye et al. [17]	2.0	- / -	-	yes
Kopp et al. [19]	2.0	- / -	-	partial
Harrer et al. [7]	2.0	- / 211	-	yes
Our Approach	2.0	762 / 82	71	yes

In our approach, we have created 762 fault tests, i.e., test

[2]The test cases are available as part of the source code of the `BPEL2oWFN` tool at http://www.gnu.org/software/bpel2owfn/download.html - Accessed on 4^{th} of August 2014

[3]See http://forge.ow2.org/plugins/scmsvn/index.php?group_id=266 to download the tests. – Accessed on 4^{th} of August 2014

[4]While the test set encompassed approx. 130 tests in their initial publication, the test set has grown to 211 feature tests to include combinations of BPEL features as well.

[5]For more details on the comparison of the two formalizations see the accompanying technical report [20].

cases for invalid processes, and reused 82 feature tests from the 211 available feature tests in [7].

III. Approach

We base our approach on the tool betsy and the findings in [7] as well as the formalization of BPEL by Kopp et al. [19]. Our approach is outlined in Section III-A and described in detail in Section III-B. The tags, which group rules for better result analysis, are presented in Section III-C.

A. Big Picture

The big picture of our approach is shown in Fig. 1. It includes the foundation of this work in the left box and our approach in the right box. The preliminary work from [7] has been used to determine the feature conformance that acts as a foundation and requirement for our approach to determine the static analysis conformance of BPEL engines.

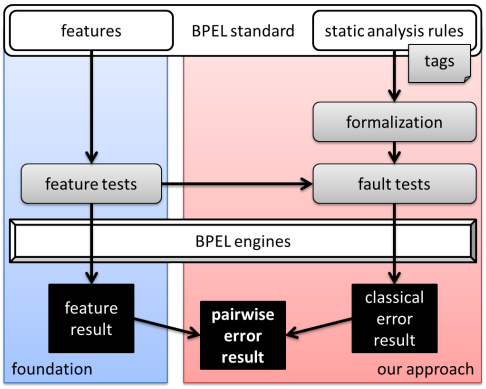

Figure 1. Big Picture of our Approach

The first step in our approach is to formalize the static analysis rules of the BPEL specification. These formalizations are either taken and adapted from [19], or have been created by the authors. With these formalizations, we can derive permutations that contain all the valid and invalid combinations of the relevant BPEL elements for this rule. Hence, we can derive test cases for the invalid combinations. This is done by selecting an appropriate feature test that includes a valid combination of BPEL elements and modify this correct feature test to contain the invalid combination, hence, creating a fault test. By doing this for every invalid combination for every rule, we create a complete test set of fault tests to be able to determine static analysis conformance. These fault tests are then used to evaluate the quality of BPEL engines, creating classical error results. Because these results may contain false positives as the engine could reject the fault test because it may contain a BPEL feature that the engine does not support, we also take feature conformance in the form of the feature results into account. This ends in the pairwise error results. As there are a lot of rules, it is hard to interpret the results per rule. Because of this, we tag the static analysis rules according to *what* BPEL feature they validate and *how* the targeted BPEL features are restricted.

B. Method

In this section, we detail three aspects, namely, the formalization[6] of the static analysis rules, the creation of the fault tests and the interpretation of the results.

The formalization is required to ensure that the resulting fault tests are complete and their numbers to not explode. Also, the quality of the tests increases as we identify minimal required changes. In this work, we propose to use the existing BPEL formalization of Kopp et al. [19] whenever possible. As they formalized the rules positively, i.e., specifying valid combinations of BPEL features, we have to negate them for creating invalid combinations which are necessary for the fault tests. When there is no formalization available, we have to model the rule ourselves. To ensure a high quality, we conduct peer review by authors and within our group, as well as compare our formalization with the one from [19]. By permuting the formalizations, we get a list of combinations with valid and invalid ones.

To create our test-pairs, we identify the required processes by features used in our formalization. In [8] we give the example of rule #47 with the following partial formalization for receiving message `<onMessage>` that receive non-empty messages:

$$[\texttt{@variable}, \text{none}] \times [\texttt{<fromParts>}, \text{none}]$$

For example to violate the rule, in this case the received message must not be completely assigned to a variable via the `@variable` attribute nor any of its parts to variables via the `<fromParts>` and its `<fromPart>` elements. As feature test, we pick the shortest process from the betsy test suite that provides all necessary features, in our example this is `Pick-CreateInstance-FromParts`. Our formalization shows the necessary modifications, so that the processes of a test-pair have a minimal difference. We derive the erroneous test from the feature test, by applying the identified modifications, resulting in `NoVariable-NoFromPart-OnMessage` for our running example by deleting the `<fromParts>` element. In addition, we change a fault test that it solely violates one rule alone, same as in the example. For optimal test results, each test shall violate a single rule and have a minimal difference to its feature test, however, as a few rules are not disjunct in their validation, this optimal state cannot be guaranteed for every test. In our experiment, 6% of the fault tests violate multiple rules.

The results are obtained by determining whether a given test has been rejected or deployed for a specific engine. To ensure that the results are correct, we propose to ensure test

[6]The formalization of the 71 rules can be found in the accompanying technical report [20].

isolation. To evaluate the results, we compare the classical and our pairwise approach. Our method to evaluate the static analysis conformance is outlined in Table II. The classical approach consists of only checking the result of the fault test, being either deployed (d) or rejected (r), with deployed meaning the fault test was wrongly accepted while rejected meaning the fault test was correctly rejected.

Table II
EVALUATION OF STATIC ANALYSIS CONFORMANCE USING TEST PAIRS

		fault test	
		deployed (d)	**rejected (r)**
feature test	**deployed (d)**	not conformant	*pair-wise conformant*
	rejected (r)	not conformant	feature unsupported

This method, however, gives too much credit for engines that reject these faults for other reasons. One could go through the log files for every test and determine manually the outcome of the fault test. But instead we propose to use an automated method by pairing the fault test with a feature test, taking the correctness of the feature test into account as well. Because the fault test combines a feature test with a fault, this pairwise approach can automatically determine whether the underlying features of the process are implemented in this engine, and determine the cause of this rejection. In our proposed pairwise approach, if and only if the feature test is correct and the fault test is rejected, we can determine that this fault has been found correctly at deployment time. Thus, we get a pairwise error result that provides an accurate means to determine the static analysis conformance of BPEL engines.

C. Rule Tags

Because of the large number of rules, we have tagged them according to two groups, namely, to their *violation check* (How are the targets checked?) and to their *target elements* (What BPEL features are restricted further?). These tags allow us to gain additional insight as they group the results in a comprehensive and easy to interpret way. In this table, only the covered rules from Table III are tagged[7]. A rule is tagged at least one time per group, and can be tagged multiple times within each tag group.

The *violation check* tags describe the type of the check. The three tags with the highest number of rules, namely, *node requirement*, *choice* and *uniqueness* compensate the lax schema definition. The rules tagged with *node requirement* requires specific elements or attribute values, where the BPEL schema provides a greater choice. If a rule demands the usage of attributes and elements in specific combinations, it is tagged with *choice*. In this case, *choice* can refer to an *inclusive or* or an *exclusive or*. A more strict schema could have made these rules obsolete in the first place, e.g., by using the native schema choice mechanism instead. Rules with the *uniqueness* tag check the uniqueness of attributes or elements, e.g., the name attributes of `<variable>` definitions have to be unique per `<scope>`. The native schema mechanisms for uniqueness could have rendered these rules redundant. There are 14 rules tagged with *consistent redundancy* that deal with nodes which carry redundant information that can be derived from the context, e.g., from other attributes, elements or the location of an activity. The tag *location* is assigned to rules that restrict possible parents or ancestors of activities. Such rules are necessary due to undesired inheritance defects of the BPEL schema types, e.g., `<rethrow>` is a common activity even though it is solely applicable in the context of `<faultHandlers>`. The rules with the tag *execution instructions* instruct the BPEL engine how to execute the process, e.g., how to lookup a variable at runtime. Conformant static analysis can detect errors that occur when the execution would be performed in the correct order, but invalid execution at runtime remains unchecked by static analysis, of course. Hence, these rules can only be checked up to a certain point with static analysis. Rules are tagged with *definition resolution* if the rules require that a BPEL activity references other BPEL, WSDL or XSD elements by a qualified name (QName) or other references. Control cycles are forbidden in BPEL and the rules that describe their detection are tagged with *control cycle detection*.

The tags in the *target elements* group indicate which BPEL features are restricted further. The majority of the rules refer to activities related to the message exchanges and their required WSDL and XSD definitions, whereas only a minority restricts the structured activities.

[7]A complete tagging of all rules can be found in the accompanying technical report [20].

IV. TEST SETUP

The test setup consists of the test suite in Section IV-A with its test cases that are executed on the engines under test briefly noted in Section IV-B by means of the test tool which is described in Section IV-C. All three parts are open source and publicly available.

A. Test Suite

In this work, we have covered 71 of the 94 rules as shown in Table III, while the remaining 23 are marked as out of scope. The latter are not covered because they either are engine-specific and we focus on engine-independent static analysis conformance, or make use of expression parsing which would require us to test the expression language XPath [21] as well. Rules #56 and #77 are the exception as they are not covered because they would require an enormous amount of additional positive tests, i.e., feature tests in varying combinations of nesting multiple activities, which are not available in the current feature test set and very time consuming to create.

Table III
STATIC ANALYSIS RULES

	rules	# of rules
Covered	1-3, 5-8, 10-20, 22-25, 32, 34-37, 44-48, 50-55, 57-59, 61-72, 76, 78-93, 95	71
Out of Scope	4, 9, 21, 26-31, 33, 38-43, 56, 60, 73-75, 77, 94	23

In total we require 762 fault tests[8], which are an average of nine tests per rule. However, the actual amount varies among the rules due to the different amount of features that need to be checked. Rule #3 has the most fault tests with 342 whereas 26 rules requires a single one. Another 33 rules have more than one fault tests but less or equal than ten, and only eleven rules have more than ten fault tests.

B. Engines Under Test

In this work, we have evaluated six different open source BPEL engines which state that they conform to the BPEL 2.0 specification and are considered mature. The engines under test comprises *ActiveBPEL v5.0.2*, *bpel-g v5.3*, *Apache ODE v1.3.5*, *OpenESB v2.2*, *Orchestra v4.9*, and *Petals ESB 4.0*.

C. Test Tool

As a test tool, we have built upon the already existing *BPEL Engine Test System* (betsy) [22], [23]. It allows evaluating whether a given BPEL process can be executed successfully onto various BPEL engines while guaranteeing test isolation. This tool has been used for or has been part of several other studies [7], [18], [24]–[26], is open source and publicly available[9]. In our work, we have used betsy to detect whether a given BPEL process can be deployed successfully on BPEL engines by checking whether the main WSDL file of the BPEL process is available via HTTP after the deployment. Moreover, for the static analysis test suite we have adapted the existing feature conformance test suite of betsy. Apart from methodical reasons, this is necessary as betsy requires specific naming conventions and allows only two specific WSDL files being the public interfaces of deployable BPEL processes.

V. RESULTS

The results[10] of the experiment described in Section IV are shown in Table IV[11]. Column-wise, the table is structured according to the six engines using the pairwise (p) approach to interpret the results of the test cases, including the delta (Δ = classic − pairwise) to show any differences between these two approaches, and the average values ($\varnothing$) as well as standard deviation (σ) in the last column. Row-wise, we present the results in a more general way using three different static analysis conformance metrics in the bottom as well as in a more detailed way per tag within their tag group, which we describe in Section III-C, at the top of the table.

The table reveals a high variation in the support for static analysis checks. **ActiveBPEL** shows full support, **OpenESB** shows no support, and the remaining four engines vary widely in their support for these static analysis rules of the BPEL specification. Consequently, the highest and the lowest rank are already set, so we concentrate on the other four engines.

The results for **bpel-g** are independent from the used approach. It has its strengths in detecting all violations regarding *execution instructions* and *definition resolution*, while its minor weaknesses are only 50% support for detecting *control cycles* and 67% support for revealing *uniqueness* errors. When looking at the BPEL activities that the rules target, bpel-g handles the *partner link*, the *variable* and the *event handler activities* perfectly, while its issues are related to *XSD definitions*, *flow activities*, and *WSDL definitions* which are only supported with a degree of 33%, 50% and 59%, respectively. A minor weakness are the *start activities* with 67%. The remaining tags in both tag systems have a very high degree of support varying around 75% or even higher. Overall, this engine performs very well as it supports 91% of the test cases, resulting in full support of 75% of the rules and in at least partial support of 86% of the rules, making it the second best engine in this regard.

Apache ODE clearly has issues with the support for the static analysis rules as it only supports 30% of the rules fully and only 51% of the rules at least partially with only a test case detection rate of 19%. The classical and the pairwise approach differ only by a few percentage points in the static analysis conformance ratings. However, when looking at the ratings of the tags, the changes vary more, ranging from 5 percentage points up to 22, but only eight tags are affected. Its major weakness is the inability to *detect control cycles*, as it is the only tag that this engine has no support at all. Moreover, *location*, *choice*, *consistent redundancy* and *node requirements* checks are only supported weakly, ranging from 17% to 37%, while the remaining ones are only supported for 47% up to 50%. Regarding the *target elements*, only the *message assignment*, *partner link*, *variable* and *correlation activities* are supported with at least 50%, while the other tags have a (considerably) lower support percentage. Overall, this engine ranks fifth when using the classical approach, but when looking at the numbers for the pairwise approach, it ranks fourth.

Orchestra places itself in the middle of the field at rank three. It has the same major weakness as Apache ODE, namely, it cannot *detect control cycles*. Regarding the other *violation checks*, support is quite high with 72% up to 75% for *choice*, *uniqueness*, *execution instructions*, and *definition resolution*, while the other three have a support rating of 42% or 50%, except for the tag *location* which has only 13% in the

[8]The tests are available at https://github.com/uniba-dsg/soca2014/tree/master/fault-tests.

[9]See https://github.com/uniba-dsg/betsy/tree/soca2014.

[10]The csv files can be found at https://github.com/uniba-dsg/soca2014/tree/master/results

[11]The atomic values can be found in the accompanying technical report [20].

Table IV
CONFORMANCE RESULTS OF OUR PAIRWISE (P) APPROACH AND THE DELTA (Δ) TO THE CLASSICAL APPROACH

	ActiveBPEL		bpel-g		Apache ODE		OpenESB		Orchestra		Petals ESB		∅		σ	
	p	Δ	p	Δ	p	Δ	p	Δ	p	Δ	p	Δ	p	Δ	p	Δ
violation check																
node requirements	100%	0%	79%	0%	37%	16%	0%	0%	42%	11%	11%	36%	45%	11%	35%	13%
choice	100%	0%	78%	0%	17%	5%	0%	0%	72%	6%	6%	44%	46%	9%	39%	16%
uniqueness	100%	0%	67%	0%	47%	0%	0%	0%	73%	7%	0%	47%	48%	9%	37%	17%
consistent redundancy	100%	0%	79%	0%	29%	7%	0%	0%	50%	7%	0%	57%	43%	12%	38%	20%
location	100%	0%	75%	0%	25%	0%	0%	0%	13%	37%	0%	75%	36%	19%	38%	29%
execution instructions	100%	0%	100%	0%	50%	0%	0%	0%	75%	0%	0%	100%	54%	17%	42%	37%
definition resolution	100%	0%	100%	0%	50%	0%	0%	0%	75%	0%	0%	75%	54%	13%	42%	28%
control cycle detection	100%	0%	50%	0%	0%	0%	0%	0%	0%	0%	0%	100%	25%	17%	38%	37%
target elements																
WSDL definitions	100%	0%	59%	0%	18%	14%	0%	0%	32%	4%	5%	31%	36%	8%	35%	11%
message activities	100%	0%	90%	0%	33%	15%	0%	0%	52%	0%	0%	76%	46%	15%	39%	28%
message assignment activities	100%	0%	86%	0%	50%	22%	0%	0%	57%	0%	0%	79%	49%	17%	38%	29%
process and scope	100%	0%	86%	0%	20%	0%	0%	0%	43%	14%	0%	36%	42%	8%	39%	13%
FCT handler activities	100%	0%	83%	0%	20%	0%	0%	0%	25%	25%	0%	42%	38%	11%	39%	17%
flow activities	100%	0%	50%	0%	8%	0%	0%	0%	50%	0%	0%	100%	35%	17%	36%	37%
partner link activities	100%	0%	100%	0%	50%	0%	0%	0%	67%	0%	11%	56%	55%	9%	39%	21%
variable activities	100%	0%	100%	0%	50%	11%	0%	0%	56%	11%	0%	33%	51%	9%	41%	12%
XSD definitions	100%	0%	33%	0%	14%	0%	0%	0%	33%	17%	17%	0%	33%	3%	32%	6%
assignment activities	100%	0%	83%	0%	14%	17%	0%	0%	17%	16%	0%	67%	36%	17%	40%	24%
correlation activities	100%	0%	80%	0%	78%	0%	0%	0%	20%	0%	0%	40%	46%	7%	41%	15%
event handler activities	100%	0%	100%	0%	14%	0%	0%	0%	80%	0%	0%	80%	49%	13%	45%	30%
loop activities	100%	0%	75%	0%	33%	0%	0%	0%	75%	0%	25%	25%	51%	4%	35%	9%
start activities	100%	0%	67%	0%	33%	0%	0%	0%	100%	0%	33%	34%	56%	6%	37%	13%
static analysis conformance																
detected rules	100%	0%	75%	0%	30%	6%	0%	0%	54%	10%	4%	52%	44%	11%	36%	19%
(partially) detected rules	100%	0%	86%	0%	51%	6%	0%	0%	79%	3%	21%	55%	56%	11%	36%	20%
detected tests	100%	0%	91%	0%	19%	4%	0%	0%	66%	1%	10%	56%	48%	10%	40%	21%

pairwise approach. Orchestra is the only engine in addition to ActiveBPEL that supports all *start activities* checks, and has at least partial support for every *target elements* tag. The *loop* and *event handler activities* are handled quite well with 75% and 80%, whereas *assignment*, *FCT handler* and *correlation activities* are supported the worst with at most 25%. Half of the tags are independent of the used approach, whereas the other half has minor (starting with 4%) as well as major (up to 37%) changes in support.

Petals ESB ranks fourth when using the classical approach, however, when using the more detailed and sophisticated pairwise approach it falls down to the fifth rank. This is because in the classical approach, Petals ESB supports 56% of the rules fully and 76% of them at least partially by detecting 66% of all test faults. However, when using the pairwise approach, it merely supports 4% of the rules fully and 21% at least partially by detecting only 10% of all test faults. Overall, support is very low. Only seven tags out of the 22 tags have support at all, ranging from 5% up to 33% in the best case. Petals ESB only supports *choice* and *node requirements* checks for *start*, *partner link* and *loop activities* as well as *XSD* and *WSDL definitions*. The results of the three tags *execution instructions*, *control cycle detection* and *flow activities* are completely inverted as the classical approach would have stated full support whereas the pairwise one states no support. Moreover, twelve other rules have been partially supported in the classical approach and have been turned to no support in the pairwise approach as well. Only the support for the tag *XSD definitions* is independent of the used approach, the results of all other 21 tags are improved using the pairwise approach.

Table V
CONFORMANCE RANKING AND PERCENTAGE

	pairwise		classic		feature	
ActiveBPEL	1	100%	1	100%	1	**82%**
bpel-g	2	75%	2	75%	**1**	**82%**
Orchestra	3	54%	3	**64%**	**5**	**45%**
Apache ODE	4	30%	**5**	**36%**	**3**	**65%**
Petals ESB	5	4%	**4**	**56%**	**6**	**24%**
OpenESB	6	0%	6	0%	**4**	**52%**

As answer to RQ1, we can state that the six engines vary greatly in terms of support for these static analysis rules. The ranking regarding the static analysis conformance of these six engines is the same whether the percentage of *detected tests*, *detected rules* or *at least partially detected rules* is used. Hence, we use *detected rules* as our main metric because it focuses on full support per rule, because in case of at most partial support we cannot call an engine rule conformant. As a result, all three metrics can be used for a ranking in our experiment. But it is not the same ranking as the one using the feature conformance as shown in Table V, as bpel-g would rank first with ActiveBPEL in the feature conformance but regarding static analysis conformance, bpel-g ranks second. Moreover, Apache ODE also loses one rank, while OpenESB drops two ranks to the last one. Both Orchestra and Petals ESB gain one rank. In the case of Petals ESB this is only

because OpenESB supports no static analysis rule at all.

VI. Discussion

This section discusses the outcomes in Section VI-A as well as threats to validity and limitations in Section VI-B.

A. Outcomes

When looking at the *violation check* tags in isolation, we can see that *control cycle detection* is implemented the least with an average of 25%, followed by the detection of the wrong *location* with 36%. Especially the lack of control cycle detection is interesting as this can result in very problematic runtime errors, and `<flow>` with `<links>` is an important part of BPEL that is used in various academic publications, e.g., the implementation of the workflow control flow patterns [27]. The other *violation check* tags are on average between 43% and 54%, indicating no general lack of support. In contrast, the support for the *target elements* tags is a bit higher as it ranges from 33% up to 56% on average. The lowest supported BPEL elements are the *XSD* and *WSDL definitions* as well as *flow* and *assignment activities* with support between 33% and 36%, while *loop*, *partner link* and *start activities* are supported the best with 51%, 55% and 56%. As every BPEL process imports and uses both XSD and WSDL definitions extensively through assignments, it seems strange that especially these activities are not well covered as part of the static analysis implementation of the engines under test. We assume this is because a few of these errors are ignored without having effects on process execution or are repaired by the engine at deploy-time by leveraging the nature of BPEL which provides redundant information in several cases.

The averages per tag are lower for the pairwise approach in contrast to the classical one. This is shown by the positive non-null values in the Δ column of the $\varnothing$ values. This is mostly caused by Petals ESB as its Δ values have the highest impact on the average. For each tag, there is a wide range of support, as can be seen by the high standard deviation ranging from 32% up to 45%, because ActiveBPEL has full support and OpenESB has none.

In summary, the pairwise approach provides more insight for three out of the six engines in this case study. For these three engines, the pairwise approach revealed lower values. While Apache ODE and Orchestra only lost on average 6 and 5 percentage points, Petals ESB lost 54 percentage points on average. This results in a change in the overall static analysis conformance ranking as shown in Table V, changing the fourth rank of Petals ESB with the classical approach to the fifth rank with the pairwise approach. Thus, we can see the effect of having a very low feature conformance on the static analysis conformance for the engine Petals ESB. In addition, we cannot see substantial effects for engines with a feature conformance of 45% or more. Thus, we have shown that the pairwise approach helps to determine the real support for the static analysis rules However, it is best for engines with a low feature conformance, thereby answering RQ2.

B. Limitations and Threats to Validity

Ensuring the quality of the tests is a critical part of this method s experiment because it directly influences the quality of the test results. While we tried to create tests that only violate a single rule, 6% of the tests, namely, 44 out of the 762 tests, violate more than a single rule. Due to this small number, their effect on the aggregated results in Table IV is small. In addition, we have used only 82 out of 211 feature tests as a base test for the fault tests. If an engine supports these 39% feature tests, our method cannot reveal any additional information, hence, in this case the classical and our pairwise approach would yield the same results. Because not every feature is restricted further in the static analysis rules, this is not problematic. What is more, 52% of the fault tests are based on four different feature tests. This, however, is not an issue as three out of the four feature tests work on every of the six open source BPEL engines, and the other one is mainly required for a single static analysis rule. Furthermore, the test suite has been kept small to minimize test explosion. Because of this, we only tested the activities themselves and cut the combinations by ignoring arbitrary nesting of activities, specifying very strict and limited nesting combinations. As we only focus on executable processes, we could ignore the concepts for abstract BPEL processes, simplifying the formalization and reducing the number of test cases required further. While the tests are executed and evaluated automatically, they have been created manually which may open the possibility for human error. An automated generation of the tests would have been very time consuming, whereas the manual creation was very straight forward. Because the tests are basically identified by the difference to their base feature test, we have kept the differences as minimal as possible and have reviewed exactly these minimal changes within the group of authors of this paper as well as within our working group. Automating the test creation is an interesting research challenge for future work. Moreover, we analyzed the test results for patterns to detect any failures in our tests as well.

After having made sure that the tests themselves are correct, we had to ensure that the results are correctly produced. For this, we have used the test isolation capabilities of betsy which provides a fresh installation of an engine for each test. Moreover, we have executed the tests three times to prevent any alterations to influence the results.

At the moment, the tests can only be executed on the BPEL engines that are supported by betsy. The experiment was not executed on the latest version for three engines, namely Petals ESB, OpenESB and Apache ODE which we have tested with the second last version. This, however, is not problematic as this work can only answer the research

questions and cannot give an accurate and always up-to-date state-of-the-art picture of these engines.

The test suite itself is not limited to open source BPEL engines, it can be used for proprietary BPEL engines as well. We did focus on the open source ones as benchmarking results of proprietary engines must be anonymized due to licensing issues, making them less interesting to publish. The created test suite is not directly applicable to other process engines, e.g., BPMN engines. Nevertheless, the method itself is portable and can be transferred to evaluate BPMN engines in the same way, but it requires the existence of a feature conformance test suite and tool.

In most cases, developers do not code the BPEL processes by hand in the XML editor but use sophisticated BPEL designers which may also include static analysis checks. In this work, however, we focus on standard conformance, or static analysis conformance to be more precise, of BPEL engines, while leaving the static analysis capabilities of BPEL designers for future work.

VII. Conclusion and Future Work

In this paper we have presented an approach to evaluate the static analysis conformance of BPEL engines. We have evaluated our proposed approach in a case study encompassing six open source BPEL engines and 762 fault tests for 71 static analysis rules. Results revealed that the static analysis conformance varies greatly between the six open source engines, ranging from no support up to full support. What is more, our proposed pairwise approach outperforms the classical one in terms of accuracy by excluding false positives. The gained knowledge can be used both by engine developers to improve their engines and by process designers to decide whether additional static analysis tools have to be used for their currently used engine.

Future work comprises evaluating BPEL designers and proprietary engines as well, covering the remaining rules that have been declared out of scope for this paper, and applying the method of this work onto other process standards and their engines, e.g., BPMN [28] and its corresponding engines.

Acknowledgements

We would like to express our gratitude to David Bimamisa and Stephan Schuberth for their initial work on part of the test cases used in this work.

References

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

[2] O. Kopp, D. Martin, D. Wutke, and F. Leymann, "The Difference Between Graph-Based and Block-Structured Business Process Modelling Languages," *EMISA*, 2009.

[3] W3C, *XML Schema (XSD)*, October 2004, v1.0.

[4] ——, *Web Services Description Language (WSDL)*, 2001, v1.1.

[5] R. Khalaf, A. Keller, and F. Leymann, "Business processes for Web Services: Principles and applications," *IBM Systems Journal*, vol. 45, no. 2, pp. 425–446, 2006.

[6] V. R. Basili and B. T. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Communications of the ACM*, vol. 27, no. 1, pp. 42–52, January 1984. [Online]. Available: http://doi.acm.org/10.1145/69605.2085

[7] S. Harrer, J. Lenhard, and G. Wirtz, "BPEL Conformance in Open Source Engines," in *Service-Oriented Computing and Applications*, 2012, pp. 1–8.

[8] C. R. Preißinger, S. Harrer, S. J. Schuberth, D. Bimamisa, and G. Wirtz, "Towards Standard Conformant BPEL Engines: The Case of Static Analysis," in *Proceedings of the 6th Central European Workshop on Services and their Composition (ZEUS 2014)*, N. Herzberg and M. Kunze, Eds., 2014.

[9] D. H. Akehurst, "Experiment in Model Driven Validation of BPEL Specifications," in *Interoperability of Enterprise Software and Applications*. Springer, 2006, pp. 265–276.

[10] J. A. Fisteus, L. S. Fernández, and C. D. Kloos, "Formal verification of BPEL4WS business collaborations," in *E-Commerce and Web Technologies*. Springer, 2004, pp. 76–85.

[11] H. Foster, S. Uchitel, J. Magee, and J. Kramer, "LTSA-WS: a tool for model-based verification of web service compositions and choreography," in *ICSE*, 2006, pp. 771–774.

[12] C. Ouyang, E. Verbeek, W. M. van der Aalst, S. Breutel, M. Dumas, and A. H. ter Hofstede, "WofBPEL: A Tool for Automated Analysis of BPEL Processes," in *ICSOC*. Springer, 2005, pp. 484–489.

[13] A. Gravel, X. Fu, and J. Su, "An Analysis Tool for Execution of BPEL Services ," in *CEC*. IEEE, 2007, pp. 429–432.

[14] N. Lohmann, "A feature-complete Petri net semantics for WS-BPEL 2.0," in *LNCS, 4th WS-FM*, 2007.

[15] OW2, "Orchestra," http://orchestra.ow2.org/, v4.9.

[16] X. Yang, J. Huang, and Y. Gong, "Defect Analysis Respecting Dead Path Elimination in BPEL Process," in *ASPCC*, 2010.

[17] K. Ye, J. Huang, Y. Gong, and X. Yang, "A Static Analysis Method of WSDL Related Defect Pattern in BPEL," in *ICCET*, 2010, pp. 472–475.

[18] S. Harrer, J. Lenhard, and G. Wirtz, "Open Source versus Proprietary Software in Service-Orientation: The Case of BPEL Engines," in *ICSOC*, 2013.

[19] O. Kopp, R. Mietzner, and F. Leymann, "Abstract Syntax of WS-BPEL 2.0," Universität Stuttgart, Tech. Rep. 6, 2008.

[20] C. R. Preißinger and S. Harrer, "Static Analysis Rules of the BPEL Specification: Tagging, Formalization and Tests," University of Bamberg, Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik no. 94, August 2014.

[21] W3C, *XML Path Language (XPath)*, November 1999, v1.0.

[22] S. Harrer and J. Lenhard, "Betsy–A BPEL Engine Test System," University of Bamberg, Tech. Rep., 2012.

[23] C. Röck and S. Harrer, "Testing BPEL Engine Performance: A Survey," University of Bamberg, Tech. Rep., 2014.

[24] S. Harrer, J. Lenhard, G. Wirtz, and T. van Lessen, "Towards Uniform BPEL Engine Management in the Cloud," in *INFORMATIK*, 2014, in press.

[25] J. Lenhard, S. Harrer, and G. Wirtz, "Measuring the Installability of Service Orchestrations Using the SQuaRE Method," in *SOCA*, 2013.

[26] J. Lenhard and G. Wirtz, "Measuring the Portability of Executable Service-Oriented Processes," in *EDOC*, 2013.

[27] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros, "Workflow Patterns," *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 5–51, 2003.

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