Improving the Static Analysis Conformance of BPEL Engines with BPELlint

Simon Harrer, Matthias Geiger, Christian R. Preißinger, David Bimamisa, Stephan J.A. Schuberth and Guido Wirtz
Distributed Systems Group
University of Bamberg
Bamberg, Germany
{*simon.harrer, matthias geiger, guido.wirtz*}*@uni-bamberg.de*
{*christian-roland.preissinger, david-chaka-basugi.bimamisa, stephan-johannes-albert.schuberth*}*@stud.uni-bamberg.de*

***Abstract*—Today, process-aware systems are ubiquitous. They are built by leveraging process languages for both business and implementation perspectives. In the typical context of a Web Services-based Service-oriented Architecture, the obvious choice to implement service orchestrations is still the *Business Process Execution Language* (BPEL). For BPEL, a variety of open source and commercial engines have emerged. Although the BPEL standard document defines a set of static analysis rules which should be checked by engines prior to deployment to be standard conformant, previous work revealed that most engines are not capable of revealing all violations of these constraints, resulting in costly runtime errors later on. In this paper, we aim to improve the static analysis conformance of BPEL engines. We implement the tool *BPELlint* that validates 71 static analysis rules of the BPEL specification, show that the tool can be easily integrated into the deployment process of existing engines, and evaluate its performance to measure the effect on the time to deploy. The results demonstrate that *BPELlint* can improve the static analysis conformance of BPEL engines with an acceptable performance overhead.**

***Keywords*-Static analysis, Standard conformance, BPEL**

I. Introduction

Nowadays, processes are ubiquitous. We use them to model our businesses, specify our implementation and even provision our cloud services [1], [2]. As a result, building such process-aware systems is critical for industry today [3]. A straight-forward means to build such systems is to define the processes using an available process language and then execute instances of the process on an available runtime, a so called process engine. One popular language for processes is the Business Process Execution Language (BPEL) [4] defined in an OASIS standard that fits very well to build service orchestrations within a Web Services-based Service-oriented Architecture (SOA) [5]. BPEL is an XML-based domain specific programming language for such orchestrations with its own syntax and semantics. To ensure that a process conforms to this language, one can use the XML schema defined by the standard to validate the syntax. But the semantics cannot be checked automatically. There is a list of rules defining static semantics of BPEL, but the rules are given in prose only. Nevertheless, the BPEL specification explicitly states that every engine that implements the BPEL standard must implement these static semantics rules [4]. We refer to this as the *static analysis conformance*. As shown in [6], the support for static analysis conformance of BPEL engines varies widely.

This is a critical issue, because when the error is not detected by the engine as part of its static analysis check, instances of the erroneous process will be executed, resulting in costly runtime errors. It is well known that the earlier an error is found, the cheaper it is to fix [7]. Because of this, we aim to improve the static analysis conformance of the BPEL engines by providing a supporting tool with the ability to evaluate the static semantics of BPEL processes. The general idea is to implement a validation tool that checks these rules as a preprocessing step as part of the deployment process [6]. Hence, we consider three requirements that our approach has to meet:

- R1 **Effectiveness** The tool MUST validate the static analysis rules of the BPEL specification.
- R2 **Integration** The tool MUST be integrable into the deployment process of an engine.
- R3 **Efficiency** The tool MUST perform its validations within an acceptable time frame.

The paper is structured as follows. It starts with related work in Section II on process languages, static analysis of BPEL, and the quality of process models. The tool *BPELlint* is described in Section III. Section IV elaborates on the fulfillment of R1 by evaluating that *BPELlint* covers 71 out of 94 of the BPEL static analysis rules, Section V fulfills R2 by providing an easy to integrate API, and Section VI fulfills R3 by providing a performance benchmark proving that *BPELlint* is usable as part of the deployment process. Finally, Section VII concludes the paper.

II. Related Work

Related work starts with the most prominent process languages to build process-aware systems in Section II-A. In Section II-B, we have a closer look at the state-of-the-art of static analysis of BPEL processes and compare the situation to that of BPMN. We conclude with an overview of assuring quality in process models in Section II-C.

A. Process Languages

Today, there is a plethora of process languages available for various purposes with specific strengths and weaknesses [1]. The three most well-known ones are the *Business Process Execution Language* 2.0 (BPEL) [4], the *Business Process Model and Notation* 2.0 (BPMN) [8] and the *XML Process Definition Language* 2.2 (XPDL) [9]. All three languages are defined as a standard, ship with execution semantics and their serialization format is XML. BPMN and XPDL are both graph-oriented and do have a visual notation, whereas BPEL is block-oriented and has none [10]. In this work, we focus on BPEL but also compare the situation of BPEL to that of BPMN, as BPMN is currently becoming more and more important and show that our approach can be adapted to further process languages.

B. Static Analysis of BPEL

This section comprises available static analysis approaches to validate BPEL processes, the role of editors in the static analysis, and the situation of the BPMN specification regarding static analysis rules.

Static Analysis Approaches of BPEL Processes: The static analysis of BPEL has been studied extensively. The typical approach is to transform the process into a formal model on which to evaluate certain conditions creating feedback on the original process model. The approaches given in Table I are categorized according to the used BPEL version, applied formalism and amount of static analysis rules of BPEL they validated. Three use Petri nets or the Petri net extension called open workflow net (oWFN), two are built upon the Eclipse Modeling Framework (EMF), another two use finite state machines (FSM), and the rest uses either a labeled transition system (LTS), the language Promela, or guarded finite state automata (GFSM). However, all except *BPEL2oWFN* do not state that they check the static analysis rules of BPEL. And *BPEL2oWFN* only checks these rules as part of the translation to open workflow nets, i.e., as a side product. *WofBPEL* does check a single rule, but at the time it was published, the static analysis rules were not defined yet, hence the authors could not have stated it explicitly. To sum up, there are a lot of different approaches of validating BPEL processes, but the validation of the static analysis rules has been neglected so far.

Editors as Static Analysis Tools: In general, BPEL processes are either automatically generated or created using a visual editor, e.g., Eclipse BPEL Designer. These editors may also implement static analysis validations. In this work, however, we focus on stand-alone tools to improve the static analysis conformance of BPEL engines that are independent of the used editor and engine.

Comparison to BPMN: The BPMN standard document [8] is at least as complex and lengthy as the BPEL specification. But in contrast, it does not comprise an extensive list of relevant checks to be fulfilled by implementing tools as it is defined in the BPEL specification. Therefore, each developer of a BPMN modeler or engine has to determine upfront, which rules have to be fulfilled in order to create or execute BPMN correctly [23]. In a recent work we have shown that this has also implications on the standard conformance of BPMN engines [24].

Table I
STATIC ANALYSIS OF BPEL PROCESSES, PARTLY TAKEN FROM [6, P. 34]

Approach	BPEL	Formalism	SA Rules
WofBPEL [11]	1.1	Petri nets	(1)
BPEL2oWFN [12], [13]	2.0	oWFN	56
Heinze et al. [14]	2.0	oWFN	
BPEL Data Flow Analyzer [15]	2.0	EMF	
BPEL Validator with OCL [16], [17]	1.1	EMF	
Yang et al. [18]	2.0	FSM	
Ye et al. [19]	2.0	FSM	
LTSA-WS [20]	1.1	LTS	
VERBUS [21]	1.1	Promela	
EA4B and WSAT [22]	1.1	GFSA	

C. Process Model Quality

Adherence to static analysis rules is not the only aspect which distinguishes the quality of process models. Moreover, research in the area of assessing and measuring the quality of models is manifold. This is due to the fact that the notion of quality can be defined in various ways and further subdivided into several aspects.

Our work on analyzing BPEL is an example for quality aspects closely related to standard conformance. For instance, a process (or tool) which fulfills all requirements stated in the official standard document is regarded as a *good* process model. In [23], [25] we presented a related approach to check the standards conformance for BPMN process models. Besides static analysis of process models, the analysis of the execution semantics is often assessed for models based on process languages as well. A common goal, regardless which process language is used, is to provide formal semantics and to guarantee the correctness of executability, especially the absence of deadlocks and livelocks (e.g., [12]–[14], [26]–[29].

Other works use well-known software quality definitions and metrics, and adapt these metrics to process languages. Often the ISO/IEC 25000 SQuaRE series [30] is the basis for these efforts. Quality is divided into various subcharacteristics herein. For example Lenhard tackles the *portability* characteristics in his works [31], [32]. Another application of this approach is the work of Mendling [33] who tries to measure and predict the error probability of processes.

Moreover there are more general approaches which provide guidelines to construct process models (e.g., [34]) or holistic approaches which take the business perspective into account (e.g., [35]).

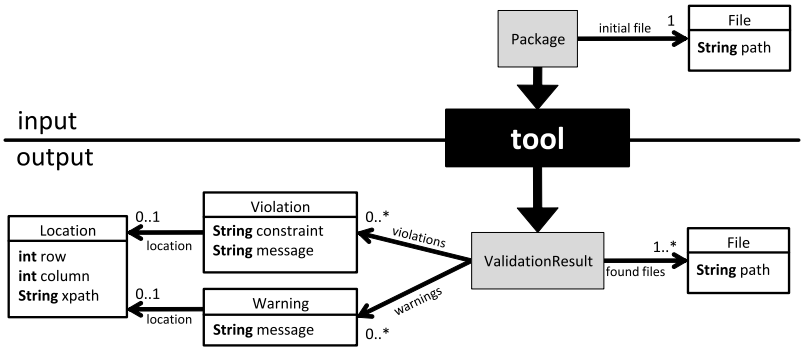

Figure 1. API of *BPELlint*

III. BPELLINT - THE BPEL VALIDATION TOOL

The BPEL specification contains 94 static analysis rules which specify the static semantics in addition to the syntax which is defined in the XML schema [4, Appendix B]. According to the specification, "conformant implementations MUST perform basic static analysis" [4, p. 13], and may even perform more advanced static analysis. However, the support for checking these static analysis rules varies greatly in a lot of engines as shown in [6]. Only a single engine out of six that have been evaluated implemented the evaluated subset of 71 rules fully. The other five covered none - or only 3, 21, 38, or 53 rules fully.

To implement these rules, i.e., to fulfill R1, we have implemented a tool called *BPELlint* that covers 71 out of 94 static analysis rules of BPEL. *BPELlint* is open source and publicly available[1]. This section describes its API (Section III-A), internal structure (Section III-B), the behavior (Section III-C) and its limitations (Section III-D).

A. API

Fig. 1 gives an overview on the proposed inputs and outputs, i.e., the API, of the tool.

Input: In the simplest cases the input is a single process file. However, processes often refer to other processes, or rely on other artifacts such as WSDL or XML Schema files. Because of this, the tool needs to access these files as well as part of a package.

Output: The output is a *ValidationResult* which comprises all findings detected during validation: It is stored which files have been analyzed in a list of *Files*. For each violation found in the course of the validation a *Violation* should be created and stored in the *ValidationResult*. Each *Violation* consists of two strings describing the violated constraint and providing a message, and a *Location* to simplify locating the issue in the affected file. The *Location* should be provided in machine-interpretable format for which we propose to use an XPath query to identify an XML node in a document, and in a human-readable format for which we propose to use a simple row/column schema. Besides violations we propose that the *ValidationResult* also comprises a list of *Warnings* which can be used to inform the user about minor issues and findings such as ignored conventions or best practices.[2]

B. Structure

In this section, we present the architecture of *BPELlint*. We structured the classes into two parts (see Fig. 2), namely the console-based user interface (*ui*) that makes use of the API and the core that implements the API in terms of the three packages.

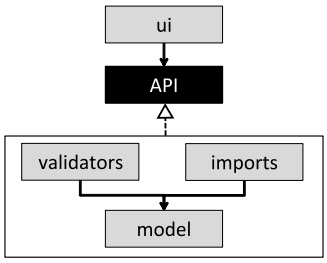

Figure 2. Package Dependencies

User Interface (API Usage): The *ui* package implements a simple command-based user interface on top of the API defined in Section III-A. *BPELlint* accepts the path to a BPEL file which acts as a starting point for a check. Moreover, you can also pass in a folder. In this case, *BPELlint* will search for every BPEL file in this folder recursively and start a check for each found one. We do not pass in a whole package as specified in the API, as on a local file system, creating a zip file is not necessary. The output is printed onto the console. Each line of the output represents either a violation or a warning. This scheme is easily understandable for both, humans and machines.

[1]For further information, have a look at the project homepage at https://github.com/uniba-dsg/BPELlint

[2]The resulting Java design can be found at https://github.com/uniba-dsg/BPELlint in the path `src/main/java/api`

Application Logic (API Implementation): The application logic comprises three packages, namely, the package *imports* that loads all files into the *model* and the package *validators* that performs the actual validation by using the *model*. The internal structure of these packages is shown in Fig. 3 and now detailed in the following.

Package imports: The *imports* package consists of classes to load the BPEL file along with its imported other XML files. Its logic is detailed in Section III-C.

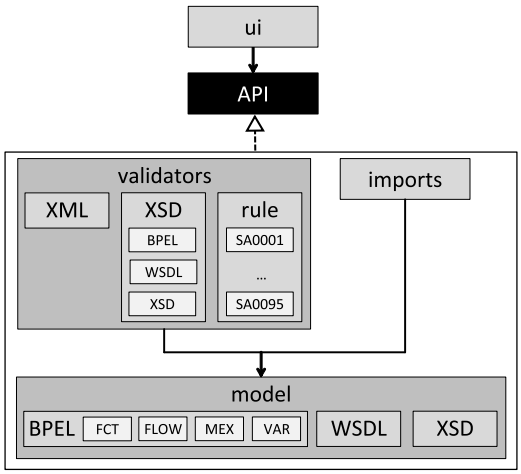

Figure 3. More detailed Package Diagram of *BPELlint*

Package model: The *model* package is a thin read-only layer over the XML structure of a BPEL process and all its required and imported documents. It is subdivided into three sub packages, namely, *BPEL*, *WSDL* and *XSD*, each of them containing the classes related to their file type. By far the largest package is the BPEL one, as it contains a class for every BPEL activity. To group these classes, we have come up with four categories. The BPEL features are separated into message exchange activities (*mex*), fault-, compensation- and termination-handlers (*fct*), variable concerning features (*var*), and flow elements (*flow*). Other BPEL model classes that did not fit in the subpackages are directly located in the *BPEL* package.

The major advantage of this model is that it implements and facilitates navigation logic. For instance, when you have an *invoke* activity, you can easily navigate to the corresponding *WSDL operation* and even to the *XSD definition* of the message that is to be sent. This allows for clear and concise rule definitions to be used during validation and enables reuse.

Package validators: There are three different types of validators, namely, the XML validators, the XSD validators and the rule validators. The XML validators check whether a file is an XML file and that it is well-formed. The XSD validators build upon that and only verify if the file is valid to its XSD schema. We have built three different XSD validators, for BPEL, WSDL and XSD files. The rule validators are

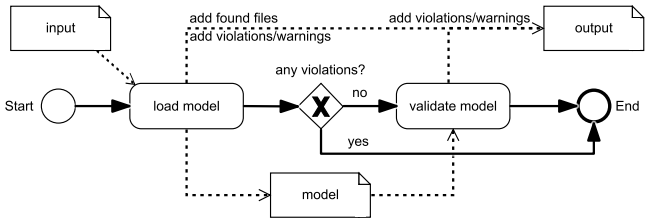

Figure 4. Bird-eyes view on the behavior of *BPELlint*

the core of *BPELlint*, as we have built one for each of the covered static analysis rules of the BPEL specification. They are built using the capabilities of the *model* enhanced with additional queries that are unique to the rule. When any of these validators finds a violation, this violation is added to the output of the API call.

C. Behavior

In addition to the structure of *BPELlint*, we present its behavior in this section. A high-level view is given in Fig. 4 as a BPMN [8] process. It shows that the logic consists of two main steps. The step *load model* creates the *model* and adds the visited files to the output whereas *validate model* uses the *model* and checks it for any rule violations. In both steps, violations and warnings can be found and added to the *output*. The only difference is, that if there are violations already after the *load model* step, the *validation model* step will not be executed. In the following, both steps are described in more detail.

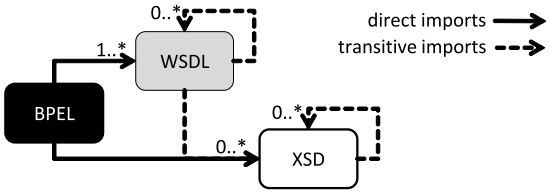

Figure 5. BPEL, WSDL and XSD Import Graph

Load Model: Loading and creating the model works on the basis of the BPEL import graph which is shown in Fig. 5. The nodes are either BPEL, WSDL or XSD files whereas the edges are import relationships. The start node is the BPEL node on the left. A BPEL process must import at least a single WSDL file and can import XSD files. WSDL files can import other WSDL files and XSD files, while XSD files can only import other XSD files. As the static analysis rules of BPEL differentiate between files that are directly imported in BPEL and files that are transitively imported, we outlined the difference in the import graph as well. In this step, each file is visited to check for three things: 1) the file exists, 2) the file is an XML file and is well-formed, and 3) the file is valid against its XML schema. If any of these conditions is not met, the model cannot be created correctly and a violation

will be created. As the validation step makes the assumption about the model that every file can be found and that it is valid to its XML schema, the validation step is not performed if any violation has been created during the *Load Model* step. Therefore, if any violations have been collected after visiting every node in the import graph, *BPELlint* immediately returns these violations as its output and the validation terminates.

Validate Model: The BPEL specification defines 94 static analysis rules[3] in [4, Appendix B]. Similar to [6], we only cover a major subset (71 out of 94) of the static analysis rules. To cover the remaining rules, we would have to take engine-specific extensions as well as the evaluation of XPath expressions into account. The evaluation of expressions refers to XPath expressions directly defined in the original BPEL process to model the data flow. Evaluating such expressions statically to determine whether the results have an expected type is rather hard.

The validation implementation in Java uses a customized version of the XML library XOM[4] v1.2.7. We added the ability to determine the row and column of every XML node. This allows us to provide location information for warnings and violations. The XPath expression to determine the XML node is computed based on the ancestors.

To implements the rules, we analyzed the rules in detail and derived a *rule specification* on which the Java implementation is based on. The *rule specification* consists of the different violation types of a specific rule and a procedural listing on how to determine any violations. This listing was used to gain an understanding how to validate the rule, and acted as a starting point for the implementation.

D. Limitations

Our tool has a few limitations. First, as described above, not every rule is covered. The rules that cover XPath expressions in the process itself and vendor-specific extensions are neglected. Second, due to the XML schema validations, we cannot cover BPEL processes with proprietary extensions violating the standard XSD, as our tool does not know these extensions.

IV. Effectiveness Evaluation of BPELlint

In this section, we check whether *BPELlint* is effective, fulfilling R1. The central idea is to evaluate the tool with test data that is sufficiently complete to determine any false negatives and false positives.

A. Test Data and Method

The *test data* consists of multiple test sets as shown in Table II. The valid processes can be used to determine that *BPELlint* does not find false positives, while the erroneous processes can be used to ensure that there are no false negatives. For the erroneous processes, we have to determine upfront which rules they violate. The main test set is that of betsy which consists of 211 valid[7] processes and 762 erroneous processes. Both valid and erroneous processes have been created methodically to ensure that they cover both the feature combinations and any rule combination [6], [39], [40]. This allows reasoning that if *BPELlint* is valid according to this test set, it effectively implements the validation. To have an even larger set of tests, we added three additional test sets as well as shown in Table II. The test assertions for the erroneous processes, i.e., the violated rules, have already been determined for the betsy tests as part of [6], so we only had to do this for the Orchestra test set by hand. In fact, *BPELlint* passes all the tests in our test sets correctly, hence, can be considered to be implement the covered rules effectively.

Table II
Test Data

Type	Valid Processes	Erroneous Processes
betsy[5]	211	762
Orchestra[6]	4	92
MEDIUM taken from [36]	2	-
LARGE taken from [37]	2	-

B. Unused Test Data in Related Work

As shown in [6, Table I], there are additional test sets used in other approaches. But [11], [17] use BPEL 1.1 processes, which are unsuitable for our BPEL 2.0 validator. The tool *BPEL2oWFN* [13] has 66 BPEL 2.0 processes in total, however, nine do not import any WSDL file, 53 do not conform to the BPEL 2.0 namespace, one is no well-formed XML file and last two were just not importable. Only a single process was checkable, and valid. Consequently, these tests could not be used as well.

C. Threats to Validity

Regarding threats to validity, the test set to evaluate the correctness of the implementation is large, but cannot contain every BPEL activity in any location, as this would result in an infinite number of test cases. This is an issue of completeness of the test data, which has already been discussed in [6], [39], [40] which have created the used betsy test set. In addition, we created the test assertions of the erroneous processes of the orchestra test set by hand. This may introduce human error, however, as the main test is already considered complete, the Orchestra test set is merely used for a secondary evaluation to strengthen the primary one with third-party processes that not have been created by at least one of the authors. Hence, any errors would not cause the evaluation to miss its goal.

[3]The rules are numbered from 1 to 95, but rule 49 is missing

[4]The customized version of XOM can be found at https://github.com/uniba-dsg/xom which extends http://www.xom.nu/.

[5]The tests can be found at https://github.com/uniba-dsg/betsy

[6]The tests can be found at http://forge.ow2.org/plugins/scmsvn/index.php?group_id=266

[7]The valid test cases have been used and described in [38]–[40]

V. Engine Integration Evaluation of BPELLint

To integrate *BPELlint* in the deployment process of available BPEL engines, i.e., to fulfill R2, the tool needs to fit both technically and functionally.

Every available open source as well as proprietary BPEL engine we know of is implemented in the Java programming language. The same is true for most available BPMN engines. Hence, as our tool is also implemented in Java, *BPELlint* can be used natively, resulting in straight-forward integration from a technical point of view.

In terms of functionality, the tool needs to answer and output the basic question of whether a checked process is valid or not. I.e., the engine can decide whether a process should be rejected during deployment or not. However, in order to facilitate debugging of invalid processes two further questions should be answered:

1) If the process is invalid, what problems occurred? and
2) What are the exact locations of the problems?

Therefore we have designed the API as described in Section III-A to determine the correctness of the process and to answer these questions. Moreover, as the API is process language independent, it can be reused for validating processes for different languages as well.

VI. Efficiency Evaluation of BPELLint

In this section, we check whether our tool fulfills R3, namely, that the checks are done efficiently to avoid a major overhead. This is important because the tool is aimed to be integrated as a gatekeeper for the deployment. And the deployment is in many cases part of a larger build chain. Hence, the longer the tool takes to check the process, the longer the deployment process and the whole build and release process will be. To verify the efficiency, we evaluate three different performance characteristics of *BPELlint*: 1) the absolute execution time on a typical developer machine, 2) the influence of the size of the BPEL process on time complexity, and 3) the influence of the number of imported WSDL and XSD files on time complexity. BPEL engines are normally run on high-end servers that have fast hardware. Hence, evaluating *BPELlint* on a developer machine marks an upper bound, i.e., the worst case, in terms of absolute execution time. Regarding time complexity, linear relations are considered efficiently. In the following, we check whether *BPELlint* is efficient regarding the definition above.

A. Benchmark

In this section, we describe how the benchmark is done by specifying the workload that is executed with the test tool that uses a specific method in a specified environment.

Environment: The benchmark is executed on a machine with an i7-3520M@2.90GHz processor with four cores, 8 GB DDR3-800 RAM and a LiteOn IT LCT-256M3S SSD. Software-wise, we use Windows 7 64bit and start the benchmark directly from within IntelliJ IDEA 14.

Test Tool: The test itself is written as a JUnit[8] v4.11 test that makes use of JUnitBenchmarks[9]. JUnitBenchmarks allows performance benchmarking of Java code. With this tool, we can specify the number of rounds the benchmark should be repeated to get stable results which are presented as the average and standard deviation of the test execution time. As the Java virtual machine takes time to warm-up, i.e., load classes or do an on-the-fly compilation to machine code, we can specify a specific number of rounds which execution times are discarded.

Workload: Our benchmark uses a workload consisting of eleven different processes as shown in Table III, which differ in their size of the BPEL process and the number of imported WSDL and XSD files. The processes are numbered using the first elven letters of the alphabet. Three are taken from the betsy test set, one from the BPEL specification [4], another three from the example processes of the OpenESB[10], and the last four from either [36] or [37]. They range from very simple BPEL processes receiving and replying only a single message up to very large ones that are hardly manageable by human developers. The size of the BPEL process is measured in terms of the number of XML elements and attributes, a measure taken from [41]. This metric is more precise than measuring the number of lines, and allows measuring the complexity of XML documents. With these processes, we are able to evaluate the relation between processing time and process size and number of imported documents.

Table III
The different workloads

ID	Source	BPEL nodes	WSDL files	XSD files
A	betsy	42	1	0
B	[4, Sec. 11.6.4]	166	1	0
C	betsy	59	2	0
D	betsy	72	1	1
E	OpenESB	368	4	1
F	OpenESB	268	4	1
G	OpenESB	187	3	2
H	[36]	1080	7	7
I	[36]	1378	9	7
J	[37]	4792	10	8
K	[37]	4938	12	8

Test Method: To ensure that the results are valid, we benchmark each of the eleven processes 110 times, subtracting the first 10 runs to take the warm-up time of the JVM into account. Thus, we get 100 rounds of data, for which we calculate the average time and the standard deviation. Moreover, we disable garbage collection during the test run, allowing the garbage collection to run only in between tests to not mess with our results. In addition, we disabled logging completely.

[8]See http://junit.org/ for further information.

[9]See http://labs.carrotsearch.com/junit-benchmarks.html v0.7.2 for further information.

[10]See http://www.open-esb.net/

B. Results

The results of the performance benchmark are shown in Table IV. For each of the workloads, we have measured and plotted the average runtime and the standard deviation in seconds. The relative standard deviation is the ratio between the standard deviation and the average, showing how much the value is alternating. Because the time values are quite small, the relative standard deviation is high for five processes (B,C,E,F,G), but for the remaining six processes the value is in an acceptable range again (between 0% and 7%). Hence, we consider the values stable enough, and work with the average in this chapter only.

Table IV
PERFORMANCE BENCHMARK RESULTS IN SECONDS

ID	average	standard deviation	relative standard deviation
A	0.01	0	0%
B	0.03	0.01	33%
C	0.05	0.02	40%
D	0.02	0	0%
E	0.12	0.05	42%
F	0.08	0.03	38%
G	0.07	0.03	43%
H	0.15	0.01	7%
I	0.19	0.01	5%
J	0.66	0.04	6%
K	0.80	0.04	5%

Absolute Time: As it can be seen in the second column in Table IV, the average time is at most 800 milliseconds for the largest workload. Our benchmark environment is a laptop, whereas BPEL engines normally run on dedicated servers. Within a similar benchmark environment, the deployment times of six process engines varied between 2.5 seconds and 18.2 seconds [42]. Putting these numbers in relation, the overhead for validating the processes A to I ranges between 0.05% and 7.6% depending on the used engine. But only if the really large processes J and K are deployed on the fastest engines a relevant impact on the deployment time occurs (32% prolonged deployment time). As a result, we can say that our implementation is efficient in most cases - and also in the worst case scenario tested the deployment time has risen only by one third. Therefore, the absolute numbers do not cause an impact on deployment as the numbers should drop even more for these workloads in production environments. Even when the server shares some resources, *BPELlint* only requires a single core.

Relation Between Time and BPEL Elements: Next, we aim to investigate the time complexity of *BPELlint*. To achieve this, we relate the average time to the number of BPEL elements as shown in Fig. 6. The graph shows that there is a *linear* relationship (i.e., $O(n)$) between the time and the number of BPEL elements. It is a very strong relationship as the R^2 value is 0.9825, hence, almost 1.0 which would be a perfect match.

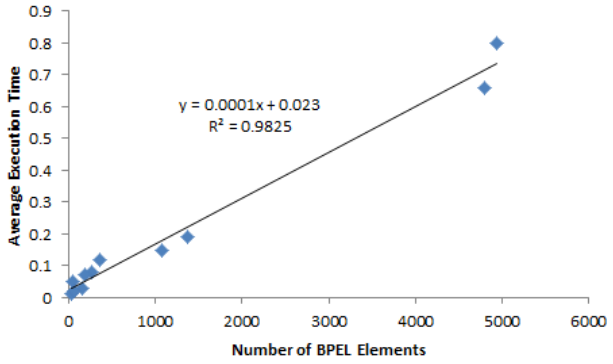

Figure 6. Relationship between average execution time in seconds and number of BPEL elements

Relation Between Time and Imported Files: In addition to the relationship between average execution time and the number of BPEL elements, the relationship between the average execution time and the number of imported files is interesting as well. A correlation graph is shown in Fig. 7. The data clearly shows an *exponential* relationship (i.e., $O(c^n)$) between the number of files and the execution time. Hence, the more imported files, the longer *BPELlint* takes to validate a BPEL package. It is a high relationship as the R^2 value is 0.8346, but not as strong as the relationship with the number of BPEL elements.

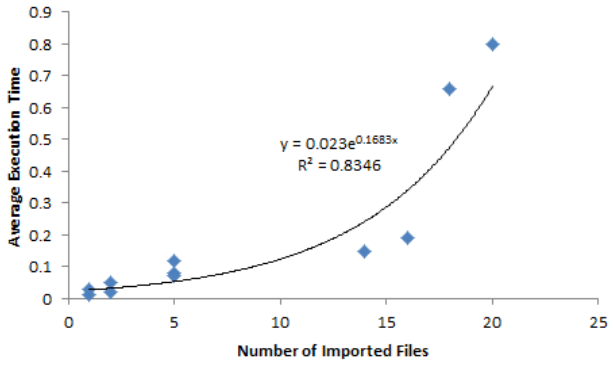

Figure 7. Relationship between average execution time in seconds and number of imported files

Summary: The results show that *BPELlint* can be integrated into the deployment process as a pre-processing step with only a minor impact on the time to deploy. One needs to be aware that *BPELlint* is slowing down the more files the BPEL file imports. This may be a limiting factor, however, a process with 20 imported files still is checked in at most 0.8 seconds on our development machine. Because of this, an influence on a high-performance machine in production is neglectable in a long continuous deployment chain.

C. Limitations

We did not measure memory consumption. This is intended, as we only require this tool for deployments, and afterwards, the memory can be freed again. Only the time is critical, as it will be part of any build or deployment process.

D. Threats to Validity

We did only benchmark our tool with correct processes. This, however, is a feature as well, because a positive process will invoke more code than an erroneous one, as in case of violations during the *Load Model* step, the tool returns earlier by skipping further validations.

VII. Conclusion and Future Work

In this work, we have postulated three requirements that have to be fulfilled by a tool that is intended to improve the static analysis conformance of BPEL engines. Moreover, we have built an API and a tool called *BPELlint* that fulfills these three requirements: the tool implements 71 out of the 94 static analysis rules of the BPEL specification, the Java API enables the tool to be integrable into existing BPEL engines, and the validation is done efficiently within at most 0.8 seconds on a typical developer machine, having only a minor impact on the whole deployment step. Therefore, *BPELlint* can be put as a preprocessing step in the deployment process to effectively and efficiently improve the static analysis conformance of existing BPEL engines.

Future work comprises the implementation of the remaining 23 static analysis rules, the performance improvement of the time complexity regarding the number of imported files, and the integration of our tool in BPEL engines. Besides, we are planning to use the presented process-language independent API to check and improve the static analysis conformance of BPMN processes as well.

References

[1] H. Mili, G. Tremblay, G. B. Jaoude, E. Lefebvre, L. Elabed, and G. E. Boussaidi, "Business Process Modeling Languages: Sorting Through the Alphabet Soup," *ACM Comput. Surv.*, vol. 43, no. 1, pp. 4:1–4:56, December 2010.

[2] T. Binz, G. Breiter, F. Leymann, and T. Spatzier, "Portable Cloud Services Using TOSCA," *IEEE Internet Computing*, vol. 16, no. 03, pp. 80–85, May 2012.

[3] M. Dumas, W. M. P. van der Aalst, and A. H. M. ter Hofstede, *Process-Aware Information Systems: Bridging People and Software Through Process Technology*. Wiley, 2005.

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

[5] C. Peltz, "Web services orchestration and choreography," *Computer*, vol. 36, no. 10, pp. 46–52, October 2003.

[6] S. Harrer, C. Preißinger, and G. Wirtz, "BPEL Conformance in Open Source Engines: The Case of Static Analysis," in *Proceedings of the 7th IEEE International Conference on Service-Oriented Computing and Applications (SOCA'14)*. IEEE, 17–19 November 2014, pp. 33–40.

[7] 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

[8] ISO/IEC, *ISO/IEC 19510:2013 – Information technology - Object Management Group Business Process Model and Notation*, November 2013, v2.0.2.

[9] WfMC, *XML Process Definition Language*, August 2012, v2.2.

[10] O. Kopp, D. Martin, D. Wutke, and F. Leymann, "The Difference Between Graph-Based and Block-Structured Business Process Modelling Languages," *Enterprise Modelling and Information Systems Architectures*, vol. 4, no. 1, pp. 3–13, 2009.

[11] 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 *Service-Oriented Computing-ICSOC 2005*. Springer, 2005, pp. 484–489.

[12] N. Lohmann, "A feature-complete Petri net semantics for WS-BPEL 2.0 and its compiler BPEL2oWFN," 212, HU Berlin, Tech. Rep., August 2007.

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

[14] T. S. Heinze, W. Amme, and S. Moser, "Compiling More Precise Petri Net Models for an Improved Verification of Service Implementations," in *Proceedings of the 7th IEEE International Conference on Service-Oriented Computing and Applications (SOCA'14)*. IEEE, 17–19 November 2014, pp. 25–32.

[15] O. Kopp, R. Khalaf, and F. Leymann, "Deriving Explicit Data Links in WS-BPEL Processes," in *Proceedings of the International Conference on Services Computing, Industry Track, SCC 2008*. IEEE Computer Society, 2008, pp. 367–376.

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

[17] ——, "Validating BPEL Specifications using OCL," University of Kent, Computing Laboratory, Tech. Rep. 15-04, August 2004. [Online]. Available: http://www.cs.kent.ac.uk/pubs/2004/2027

[18] X. Yang, J. Huang, and Y. Gong, "Defect Analysis Respecting Dead Path Elimination in BPEL Process," in *IEEE Asia-Pacific Conference on Services Computing (APSCC) 2010*, 2010, pp. 315–321.

[19] K. Ye, J. Huang, Y. Gong, and X. Yang, "A Static Analysis Method of WSDL Related Defect Pattern in BPEL," in *IEEE International Conference on Computer Engineering and Technology (ICCET) 2010*, 2010, pp. 472–475.

[20] H. Foster, S. Uchitel, J. Magee, and J. Kramer, "LTSA-WS: a tool for model-based verification of web service compositions and choreography," in *Proceedings of the 28th international conference on Software engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 771–774. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134408

[21] 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.

[22] A. Gravel, X. Fu, and J. Su, "An Analysis Tool for Execution of BPEL Services ," in *E-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, 2007. CEC/EEE 2007. The 9th IEEE International Conference on*. IEEE, 2007, pp. 429–432.

[23] M. Geiger and G. Wirtz, "BPMN 2.0 Serialization - Standard Compliance Issues and Evaluation of Modeling Tools," in *5th International Workshop on Enterprise Modelling and Information Systems Architectures*, St. Gallen, Switzerland, September 2013, pp. 177–190.

[24] M. Geiger, S. Harrer, J. Lenhard, M. Casar, A. Vorndran, and G. Wirtz, "BPMN Conformance in Open Source Engines," in *Proceedings of the 9th IEEE International Symposium on Service-Oriented System Engineering (SOSE)*. San Francisco Bay, USA: IEEE, March 30 – April 3 2015, (to appear).

[25] M. Geiger and G. Wirtz, "Detecting Interoperability and Correctness Issues in BPMN 2.0 Process Models," in *ZEUS, Rostock, Germany, February 21-22, 2013*, ser. CEUR Workshop Proceedings. CEUR-WS.org, Feb 2013, pp. 39–42.

[26] R. M. Dijkman, M. Dumas, and C. Ouyang, "Semantics and analysis of business process models in BPMN," *Information and Software Technology*, vol. 50, no. 12, pp. 1281–1294, 2008.

[27] P. V. Gorp and R. Dijkman, "A visual token-based formalization of BPMN 2.0 based on in-place transformations," *Information and Software Technology*, vol. 55, no. 2, pp. 365 – 394, 2013, special Section: Component-Based Software Engineering (CBSE), 2011.

[28] 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.

[29] W. van der Aalst, "Workflow verification: Finding control-flow errors using petri-net-based techniques," in *Business Process Management*, ser. Lecture Notes in Computer Science, W. van der Aalst, J. Desel, and A. Oberweis, Eds. Springer Berlin Heidelberg, 2000, vol. 1806, pp. 161–183.

[30] ISO/IEC, *ISO/IEC 25000:2014; Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE)*, 2014.

[31] J. Lenhard and G. Wirtz, "Measuring the Portability of Executable Service-Oriented Processes," in *Proceedings of the 17th IEEE International EDOC Conference*. Vancouver, Canada: IEEE, September 2013, pp. 117 – 126.

[32] ——, "Detecting Portability Issues in Model-Driven BPEL Mappings," in *SEKE*, Boston, Massachusetts, USA, June 2013.

[33] J. Mendling, *Metrics for Process Models: Empirical Foundations of Verification, Error Prediction, and Guidelines for Correctness*, 1st ed. Springer Publishing Company, Incorporated, 2008.

[34] J. Mendling, H. A. Reijers, and W. M. van der Aalst, "Seven process modeling guidelines (7pmg)," *Information and Software Technology*, vol. 52, no. 2, pp. 127–136, 2010.

[35] S. Overhage, D. Birkmeier, and S. Schlauderer, "Quality marks, metrics, and measurement procedures for business process models," *Business & Information Systems Engineering*, vol. 4, no. 5, pp. 229–246, 2012.

[36] A. Schönberger and G. Wirtz, "Towards Executing ebBP-Reg B2Bi Choreographies," in *Proceedings of the 12th IEEE Conference on Commerce and Enterprise Computing (CEC 2010), Shanghai, China*. IEEE, November 10-12 2010.

[37] S. Harrer, A. Schönberger, and G. Wirtz, "A Model-Driven Approach for Monitoring ebBP BusinessTransactions," in *Proceedings of the 7th World Congress on Services 2011(SERVICES2011), Washington, D.C., USA*. IEEE, July 2011, pp. 61–68.

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

[39] 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. 237–244.

[40] ——, "Open Source versus Proprietary Software in Service-Orientation: The Case of BPEL Engines," in *International Conference on Service Oriented Computing*, vol. 8274. Berlin, Germany: Springer Berlin Heidelberg, 2013, pp. 99–113.

[41] 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.

[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.