Towards Standard Conformant BPEL Engines: The Case of Static Analysis

Christian R. Preißinger, Simon Harrer, Stephan J. A. Schuberth, David Bimamisa and Guido Wirtz

Distributed Systems Group, University of Bamberg, Germany
{simon.harrer,guido.wirtz}@uni-bamberg.de
{david-chaka-basugi.bimamisa,christian-roland.preissinger,stephan-johannes-albert.schuberth}@stud.uni-bamberg.de

Abstract. The errors in BPEL processes that are only detected at runtime are expensive to fix. Several modelers and process engines for BPEL exist, and the standard defines basic static analysis (SA) rules as a detection mechanism for invalid processes, but the actual conformance of BPEL modelers and engines regarding these rules is unknown. We propose to develop test cases to evaluate the conformance of BPEL modelers and engines regarding static analysis. The evaluation results enable decision makers to identify and use the most conformant engine and modeler that detect errors before runtime and therefore reduce costs.

Keywords: SOA, BPEL, static analysis, conformance testing

1 Motivation

BPEL, a standard [9] by OASIS, defines a graph and block structured process language (see [6]), corresponding execution semantics, and 94 basic static analysis (SA) rules[1]. "The purpose of [these rules] is to detect any undefined semantics or invalid semantics within a process definition that was not detected during the schema validation against the XSD" [9, p. 194]. These rules seem rather simple, but are nevertheless as important for executing BPEL processes as static type checking is for executing Java applications. Consequently, one would expect the IDEs (BPEL modelers) and the runtime (BPEL engines) to detect any violations of these rules. Especially, as the BPEL specification requires a fully standard conformant engine to implement all static analysis rules [9, p. 13].

Each static analysis rule defines constraints for at least one BPEL element, e.g., rule #47 enforces, among other conditions, that any received non-empty message must be stored in variables. For example, consider a developer implements a simple BPEL process[2] with two variables (input and output variable) which awaits a message (`onMessage`) which instantiates the process, copies the contents of the

[1] The rules are enumerated from 1 to 95, but 49 is missing, thus 94 rules exist.

[2] The process is available at `https://lspi.wiai.uni-bamberg.de/svn/betsy/sa47-test.zip` - Accessed 02/14/2014

input variable into the output variable (`assign`) and returns the output variable (`reply`), but forgets to store the received message in the designated input variable, hence violating rule #47. Against expectations, the error is neither detected by the two widely used Open Source BPEL IDEs, *Eclipse BPEL Designer v1.0.3* and the *OpenESB IDE v2.3.1*, nor by the BPEL analysis tool *BPEL2oWFN*. Moreover, only the BPEL engines *Apache ODE 1.3.6* and *bpel-g v5.3* correctly reject the erroneous process whereas *OpenESB v2.3.1* and *Orchestra 4.9* falsely accept and deploy the process. Upon execution of the process, both engines show behavior which is hard to debug: *OpenESB* returns `null` with the error of a `NullPointerException` and *Orchestra* returns a timeout with no error trace. In this particular case, none of the modelers or analysis tools and only two out of four engines were able to detect this basic error.

Thus, this preliminary evaluation raises the following research question: *What is the conformance to the static analysis rules of BPEL modelers and engines and why is this the case?*

2 Related Work

Static analysis regarding BPEL has been studied extensively in literature. Each considered approach was analyzed by investigating three aspects: the BPEL version, the number of test cases, and the amount of static analysis rules covered by the tests. The approaches [1–3, 11] focus on BPEL 1.1 whereas [7, 12, 13] 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 [1] and Ouyang et al. [11] provide 16 and 30 valid BPEL 1.1 processes as test cases, respectively. In addition, Ouyang et al. [11] presents 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, only [7] provides test cases[3]. Each of these 56 test cases corresponds to a specific static analysis rule. Moreover, Lohmann presents the tool *BPEL2oWFN* which automatically detects violations of these 56 rules as a positive side effect during a transformation from BPEL to Petri Nets [8, p. 34]. Because of a different focus of [7], the 56 provided tests are not suitable to evaluate the conformance to the static analysis rules of BPEL engines. They do not include all error types of the covered 56 rules and are abstract (no WSDL interface, incomplete process definition).

Whereas the discussed approaches check the standard conformance of BPEL processes, none of them evaluates the standard conformance of BPEL modelers or engines. Harrer et al. [4, 5] did focus on BPEL engines with their automated testing tool *betsy*, but solely evaluated standard conformance using approx. 130 valid processes. Thus, standard conformance regarding the static analysis rules of BPEL engines remains untested [4, p. 7], as is the case for BPEL modelers.

[3] 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 01/22/2014

3 Research Outline

To answer the research question, we aim to a) create test cases ($\xrightarrow{\text{derive}}$ in Fig. 1) for the 94 static analysis rules and b) use these test cases to analyze static analysis conformance of BPEL modelers and engines ($\xrightarrow{\text{evaluate}}$ in Fig. 1) with *betsy*, i.e., determining the degree of static analysis conformance.

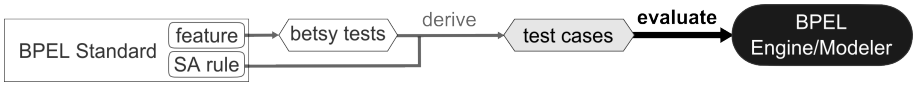

Fig. 1. Big Picture of our Approach

The derivation of the test cases for each static analysis rule is subdivided into six steps. First, the related elements and attributes are extracted from the textual representation of the rule and partitioned into groups. Second, the elements and attributes are permutated into a list of possible combinations. Third, we identify valid, invalid, and meaningless combinations according to the standard restrictions. Fourth, we calculate the distance from each invalid combination to every valid combination and pair up the least distant combinations. Our distance metric is the amount added and removed elements and attributes. Fifth, to create the test case we select the most feature-poor process from a test set of valid processes (e.g. the *betsy* test set) that fits the valid combination. Sixth, we mutate the valid process to an invalid one corresponding to the invalid combination. This approach increases the quality of the tests as it ensures that the erroneous process solely violates a single error condition. Especially the first and the third step are hard to automate as interpreting prose is nontrivial. Thus, four steps are automated whereas the other two are done manually. Following our running example, we applied the first step of our proposed procedure onto rule #47 and determined the formalization of influencing factors in the listing below. Step two to five are not shown. Regarding step six, the test case described in Sect. 1 implements the error condition represented by the permutation marked as bold.

{empty message, **non-empty message**} ×
{invoke, receive, reply, **onMessage**, onEvent}×
{**incoming**, outgoing}×
{variable assignment, **no variable assignment**}×
{part assignment, **no part assignment**}

An engine passes such a test case if the process is rejected during deployment. But there may be false positives, i.e., the process is rejected by an engine because of an unsupported feature or an internal error. To prevent misleading results, we propose to take the betsy conformance evaluation into account. betsy reveals [4, p. 6] that full standard conformance is far from given by detailing which BPEL feature is supported by which engine. To avoid false positives during the evaluation of a single error type, we suggest to use a pair of BPEL processes,

a fully functional and an erroneous one. We assume that if the engine rejects the valid process, it is not able to detect this error type. But this assumption introduces false negatives, as the engine may reject the erroneous process by static analysis and the valid one by missing feature support. To counter this, we propose to evaluate the log files for any hints on why the deployment failed. The evaluation of the BPEL modelers is analogous.

The evaluation with test pairs of valid and invalid tests reveal the quality of the error detection, making the engines and modelers comparable in this regard. As the degree of error detection has an impact on the development costs, this metric may be leveraged for buying decisions. In addition, it can also be used for improving the products and act as a regression test suite.

The requirements to use our approach are 1) a process language standard defining rules for valid processes and 2) a feature complete set of valid processes. We use the process language BPEL in this case study, but the approach is also applicable for other process languages, e.g., for the Business Process Modeling and Notation (BPMN) [10].

References

1. D. H. Akehurst. Validating BPEL Specifications using OCL. Technical Report 15-04, University of Kent, Computing Laboratory, August 2004.
2. J. A. Fisteus, L. S. Fernández, and C. D. Kloos. Formal verification of BPEL4WS business collaborations. In *EC-Web LNCS 3182*, pages 76–85. Springer, 2004.
3. H. Foster, S. Uchitel, J. Magee, and J. Kramer. LTSA-WS: A Tool for Model-Based Verification of Web Service Compositions and Choreography. In *ACM ICSE*, 2006.
4. S. Harrer, J. Lenhard, and G. Wirtz. BPEL Conformance in Open Source Engines. In *IEEE SOCA*, 2012.
5. S. Harrer, J. Lenhard, and G. Wirtz. Open Source versus Proprietary Software in Service-Orientation: The Case of BPEL Engines. In *ICSOC*, volume 8274 of *LCNS*, pages 99–113, Berlin, Germany, 2013. Springer Berlin Heidelberg.
6. 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*, 4(1):3–13, June 2009.
7. N. Lohmann. A feature-complete Petri net semantics for WS-BPEL 2.0. In *LNCS, 4th WS-FM*, 2007.
8. N. Lohmann. A feature-complete Petri net semantics for WS-BPEL 2.0 and its compiler BPEL2oWFN. Technical report, 212, HU Berlin, August 2007.
9. OASIS. *Web Services Business Process Execution Language*, April 2007. v2.0.
10. OMG. *Business Process Model and Notation*, January 2011. v2.0.
11. C. Ouyang, E. Verbeek, W. van der Aalst, S. Breutel, M. Dumas, and A. ter Hofstede. WofBPEL: A Tool for Automated Analysis of BPEL Processes. In *ICSOC*, 2005.
12. X. Yang, J. Huang, and Y. Gong. Defect Analysis Respecting Dead Path Elimination in BPEL Process. In *IEEE APSCC*, 2010.
13. K. Ye, J. Huang, Y. Gong, and X. Yang. A Static Analysis Method of WSDLRelated Defect Pattern in BPEL. In *IEEE ICCET*, 2010.