# Calculating Test Coverage for BPEL Processes With Process Log Analysis

Daniel Lübke

Leibniz Universität Hannover
FG Software Engineering
Welfengarten 1, D-30167 Hannover, Germany
Email: daniel.luebke@inf.uni-hannover.de

***Abstract*—Today more and more business processes are digitized by implementing them in specialized workflow languages like the Business Process Execution Language (BPEL) or Business Process Model and Notation (BPMN 2.0), which orchestrate services along the process flow. Because these process models are software artefacts of critical importance to the functioning of the organization, high quality and reliability of these processes are mandatory. Testing therefore becomes an important activity in the development process. Test Coverage Metrics have long been used in software development projects to assess test quality and test progress. Current approaches to test coverage calculation for BPEL either relies on instrumentation, which is slow, or is limited to vendor-provided unit test frameworks, in which all dependent services are mocked (unit tests), which limits the applicability of such approaches. Our approach relies on analyzing process event logs that are written during process execution. This approach does not require additional infrastructure and can be used in unit tests, as well as in system and integration tests. We found that our approach for measuring test coverage is not only more flexible but also faster than an instrumentation-based approach.**

***Keywords–Test Coverage; Process Mining; BPEL; Event Log.***

## I. Introduction

Executable Business Processes, implemented with WS-BPEL or BPMN2, are used to automate business processes in large companies. They are software artefacts and can contain complex orchestration logic. With the increasing demand for fully digitized solutions, it is likely that more and more business processes are being implemented in these or similar orchestration languages.

Because business processes and as such their software implementations are very critical to the functioning and performance of organizations, it is mandatory to do good quality assurance in order to avoid costly problems in production [1]. It has been shown by Piwowarski et. al [2] that a) test coverage measurements are deemed beneficial by testers, although b) they are rarely applied because of being difficult to use, and c) that higher coverage values lead to more defects being found. These findings are supported by Horgan et al. [3], who linked data-flow testing metrics to reliability, and Braind et al. [4], who simulated the impact of higher test coverage. Furthermore, Malaiya et al. [5] and Cai & Lyu [6] have developed prediction models that can link test coverage with test effort and software reliability.

Quality Assurance, and thus test coverage calculation, are an ongoing activity because executable processes will evolve over time [7]. One way for continously measuring test quality is to measure test coverage as part of all ongoing testing activities. Test Coverage then serves as measurement of test data adequacy [8].

While approaches applicable for unit testing executable processes have been proposed by academia (e.g., [9], [10]) and developed by vendors for their respective process engines, there is no practical way to efficiently calculate test coverage for tests that are not controlled by a process testing framework. Also, approaches relying on instrumentation create significant additional overhead by a factor larger than 2.0 compared with the "plain" test case execution times [11] – which is far more than instrumentation approaches for "normal" programming languages, e.g., Java, impose.

For improving the guidance of quality assurance in software projects developing executable processes, an approach is required that can be used in non-unit test scenarios and is ideally faster and easy to set up. Within this paper, we propose a new approach based on analyzing process event logs, which are written by process engines regardless of whether testing frameworks are used or not. Our research goal is to calculate test coverage metrics more flexible and faster by leveraging processes' events logs.

The paper is structured as follows: First, the process modeling language BPEL is shortly explained in Section II before related work is presented in Section III. Our approach for mining test coverage metrics is described in detail in Section IV. For validating our approach, we conducted an experiment, which is presented in Section V. Finally, we conclude the paper and give an outlook on future work.

## II. Background on BPEL

BPEL (short for WS-BPEL; Web Services Business Process Execution Language) is an OASIS standard that defines a modeling language for developing executable business processes by orchestrating Web services.

BPEL Models consist of *Activities*, which are divided into *Basic Activities* and *Structured Activities*. *Basic Activities* carry out actual work, e.g., performing data transformations or calling a service, while *Structured Activities* are controlling the process-flow, e.g., conditional branching, loops, etc.

Important *Basic Activities* include the *invoke* activity (which calls Web services), the *assign* activity (which performs data transformations), and the receive and reply activities (which offer others to call a process via service interfaces). Important Structured Activities are the *if*, *while*, *repeatUntil*, and

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-614-9 

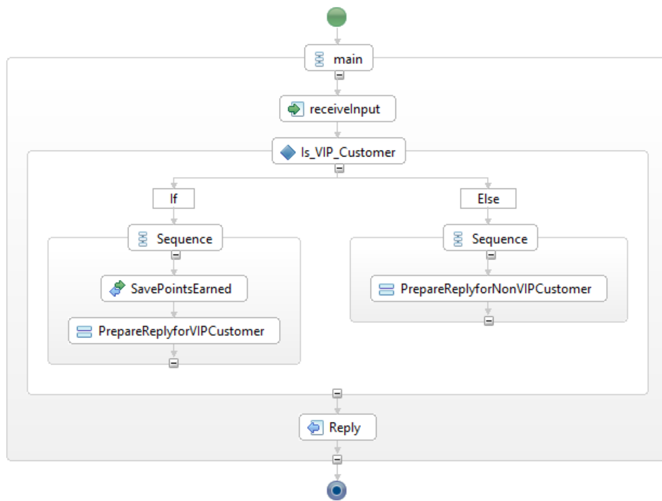


Figure 1. Sample BPEL Process for processing an Order.

*forEach* activities, which offer the same control-flow structures like their pendants in general purpose programming languages, and the *flow* activity, which allows process designers to build a graph-based model including parallel execution. For building the graph, BPEL offers *links* that can also carry conditions for modelling conditional branches.

For handling error conditions and scoped messages, BPEL provides different kinds of *Handlers*: *Fault Handlers* are comparable to try/catch constructs: Whenever a SOAP Fault is returned by an invoked service or is thrown by the process itself, the Process Engine searches for defined Fault Handlers. These may trigger *Compensation Handlers*, which can undo already executed operations. For receiving events asynchronously outside the main process flow, *Event Handlers* can be defined. These come in two flavors: *onEvent Handlers* for receiving SOAP messages, and *onAlarm Handlers* for reacting on (possibly reoccurring) times and time intervals.

BPEL does not define a graphical representation, like, e.g., BPMN2 does, but standardizes the XML format in which it is saved. Vendors have developed their own graphical representation. Within this paper we use the notation used in Eclipse's BPEL Designer. A process that will be used for examples in this paper is shown in Figure 1: A customer places an order ("receive input"). A check is made, whether the customer has VIP status or not. In case of a VIP customer, points are credited to the customer's account ("SavePointsEarned"). In both cases appropriate response message to the customer are prepared ("PrepareReplyFor…"), which is then sent back to the customer ("reply").

BPEL processes are deployed to a *Process Engine*, which has the responsibility for executing process instances and managing all aspects around process versioning, persistence, etc. The amount of data, which is persisted during process execution, is vendor-dependent and can be configured during the deployment of a process model in most engines.

BPEL has been designed to be extensible. Many extensions by both standard committees and vendors have been made. For example, BPEL4People allows to interact not only with services but also with humans during process execution.

## III. Related Work

With the rise of BPEL, testing of these critical software artefacts became subject of many research projects. For example, Li et al. (BPEL4WS Unit Testing Framework [9]), Mayer & Lübke (BPELUnit [10]), and Dong et al. (Petri Net Approach to BPEL Testing [12]) published their ideas.

The BPELUnit framework was later extended by Lübke et al. [11] with test coverage measurement support. First, the metrics needed to be defined, which is not as straightforward as for other languages due to BPEL's different mechanisms for defining the process-flow. Consequently, three metrics were defined: **Activity Coverage**, **Handler Coverage**, and **Link Coverage**. Coverage Measurement was done by instrumenting the BPEL process: For tracing the execution, the process is changed prior to deployment. Additional service calls are inserted for every activity. The service calls send the current position ("Markers") to the test framework. Because of this the test framework knows which activities have been executed in the test run. However, the test framework needs to run while the instrumented processes are executed, which makes its use limited to automatic tests only. Also, the overhead introduced by many new service calls is considerable: The reported overhead is more than 100%, i.e., the test execution times are more than doubled. This is because every execution trace point needs to be send out of the process via a service call, which requires XML serialization and involves the network stack. This also makes the BPEL process much larger: The number of basic activities tripples for instrumenting all measurement points for calculating activity coverage alone. One advantage of the approach is that is only slightly dependent on the Process Engine being used: The changes to the BPEL process are completely standards-compliant. Only the new service for collecting the Markers needs to be added to the engine-specific deployment descriptor.

Process Engine vendors have also developed their own proprietary solutions: Test Cases are developed in the development environment of the process engine and can be executed from there or on a server. All services are mocked and the test frameworks simply inject predefined SOAP messages. Such test frameworks use a striped-down version of the process engine. This results in a mixture between simulation and test: The process engine uses the same logic but not all parts of its code are triggered because some features are disabled. Also, there is no possiblity of calling "real" services instead of mocks. While test coverage calculation is very fast, because the algorithms have access to internal engine data, its use is limited to unit test scenarios only. Examples of such vendor-provided test frameworks are Informatica's BUnit [13] and Oracle's BPELTest [14].

All in all, there is currently no approach available for BPEL processes that can be used to measure test coverage on code level with acceptable performance and the ability to be used in conjunction with manual tests and integration & system tests.

## IV. Test Coverage Mining

This section will present the different steps that are performed for analyzing the process log in order to calculate test

coverage.

### A. Metric Calculation Process

For calculating test coverage, we use process mining techniques. Process Mining is concerned to build "a strong relation between a process model and 'reality'captured in the form of an event log" [15, p. 41]. By having the BPEL process model and all test cases available as event logs from the process engine, we are able to "replay" the event logs generated from the tests on top of the BPEL process model. Out of the many possible motivations to do a replay, our goal is to extend our model with frequency information [15, p. 43].

Accordingly, our approach is divided into three sub-steps, which are described in the following sections:

1) Build the BPEL Process Model Syntax Tree from its XML representation (BPEL Analysis),
2) Fetch the event log from the Process Engine (Data Gathering),
3) Replay the event logs on top of the BPEL Process and calculate coverage metrics (Mining).

### B. BPEL Analysis

Within this step, the BPEL XML representation is read and the control-flow graph is being constructed as described by the block-based structured activities. For example, activities contained in *sequence* activity are chained together by control-flow links. The construction of the control-flow graph is the same as for the instrumentation approach to measuring BPEL test coverage [11] and thus takes the same time to build. The BPEL Models are accessible via the process engine's repository and can be extracted as part of the coverage mining.

### C. Data Gathering

The case study project, which we could analyze, uses Informatica ActiveVOS [13]. ActiveVOS is a process engine fully compliant with the BPEL 2.0 and BPEL4People standards and stores all data (process models, process instances, process logs, ...) in a relation database. This allowed us to access and analyze the available data that can be mined for test coverage metrics. For different persistence settings ActiveVOS stores different lifecycle events for every BPEL activity, which include *ready to execute*, *executing*, *completed*, *faulting* and *will not execute*. In addition, there are two more event types for links (edges for graph-based modeling): *link evaluated to true* and *link evaluated to false*. Besides the event time, the event timestamp, the corresponding process instance, an internal activity or link ID is logged.

This means that all necessary data is available in order to reconstruct the process-flow and thereby calculating the test coverage metrics: For calculating activity coverage, all *completed*, *faulting* and *link evaluated* events need to be fetched for a given test run. All other event types can be ignored, which allows to use all engine settings except for "no logging." The underlying conceptual data model, as it is implemented in the ActiveVOS engine, is shown in Figure 2.

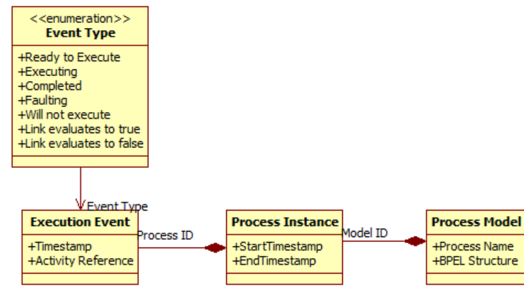


Figure 2. Conceptual Data Model of the Process Engine being used.

### D. Replay & Metric Calculation

Test Coverage can be calculated with the data extracted in the previous steps. At first, all activities and conditional links in the syntax tree are marked as not executed. In the second step, all events are being applied to the syntax tree and all activities that have a corresponding *completed* or *faulting* event are marked as being executed. Also, conditional BPEL links for graph-based modeling are marked. However, every link can carry two different markers: one if the condition was true and another if the condition was false. Because links without a condition are excluded from the coverage metric, they are ignored from further analysis.

During this phase, loop activities can be marked as executed twice for calculating the branch coverage in later stages. This happens for most loops by setting this marker, if the loop is executed twice after each other. The only exception is the parallel *forEach* loop, in which the activity ID contains the number of the currently executed parallel branch. If a counter larger than one is encountered, the *forEach* activity is marked as executed twice.

The main problem in this step was to link events and activity nodes in the BPEL model. Because the activity IDs in the events are generated by the process engine and are not part of the BPEL model, it is necessary to first resolve the proprietary IDs to the activities. The generated activity IDs are in an XPath-like structure, which closely resembles the XML structure of the BPEL model. However, some cleaning and re-writing of these IDs is necessary, because they sometimes reference internal data structures and do not directly map to the BPEL activities. After re-writing the IDs, they can be converted to XPath expressions that directly point to the BPEL activity being executed. This step is highly specific to the process engine being used and needed reverse-engineering the format and construction rules for the proprietary IDs.

After all events have been applied to the syntax tree, the coverage metrics can be calculated. The easiest test coverage metric to compute is the Activity Coverage $C_A$ metric: The syntax tree is traversed and all activities are counted which are marked ($A_m$) and which are not marked ($A_u$) as shown in equation 1.

$$C_A := \frac{|A_m|}{|A_m| + |A_u|} \tag{1}$$

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-614-9

Similarily, Handler Coverage $C_H$ can be calculated by searching the syntax tree for handlers that have been successfully executed.

Calculating Link Coverage $C_L$ by process mining is easier than with instrumentation: In order to detect the different conditions on links, instrumentation needs to insert many new links and activities. However, with process mining, dedicated events are triggered whenever a link condition has been evaluated.

### E. Example

To illustrate the replay of the event log on top of the process model we assume two test cases for the example BPEL process as shown in Figure 3. The first test case tests the VIP Customer.

As can be seen by the trace, the completion events are differently ordered than the definition in the BPEL process model: structured activities like a *sequence* or an *if* are completed after all their child activities have been completed. The replay algorithm needs to take this into account when replaying the event log against the process model.

Taking the event log for the first test case and replaying it on top of the BPEL process model yields the markings as illustrated in the center of Figure 3. Replaying the second test case yields the markings as shown on the right hand side of the same figure. With these two test cases, all basic activities are covered.

### F. Comparison to Instrumentation

When we compare our approach to instrumentation (see Figure 4), there are many parts of the calculation that are similar or even the same. Instrumentation would initially load the BPEL process model and construct a syntax tree. However, it would then change the process model by introducing service calls that signal the internal process state to the test framework. During run-time these service calls are equivalent to log events. These events are replayed on the process model in both approaches. Thus, the main differences are that

- instrumentation needs to build the syntax tree prior to the test run and a service receiving all markers must be active during the whole test while process mining can perform all activities after the test run is completed,
- instrumentation needs to change the BPEL process model while process mining does not, and
- the events are collected in the instrumentation approach by signaling service calls instead of extracting all event logs with one database query like in our approach. For a test run, the instrumentation approach requires at least as many service calls for signaling the process state as the number of executed basic activities depending on the coverage metrics that shall be calculated.

Due to these structural differences, we expect our approach to be overall faster than the instrumentation approach: Making and answering many fine-grained service calls is time-consuming as outlined above. Being able to fetch all events from the Process Engine's event log at once should yield better performance. In addition, our approach does not slow down execution times of the executable processes itself because they behave as they are implemented and are not changed by an instrumentation process and their run-time behavior is not altered by introducing probes. This means that no additional error sources (e.g., by defects in the instrumentation) or different behavior (e.g., in parallel activities by instrumentation code) can occur.

### G. Sample Implementation

We implemented a tool that performs the outlined test coverage calculation. The tool connects to the database of the process engine and extracts all relevant information. After the tests have been completed, the tool extracts the events for all newly created process instances. It expects that the tested processes have been configured appropriately to at least store the events for successfully and unsuccessfully completed activities.

The implementation is highly dependent on the process engine being used. The available process log data and its format is defined by vendors because it is not specified in any standard. As outlined in the previous section, a post-processing of the event log data is needed in order to properly resolve the referenced activities.

## V. Experiment

In order to evaluate our approach, we conducted a small experiment that is described in this section.

### A. Experiment Description & Design

For evaluating the practical applicability and the performance implications of our approach, we want to research the following two research questions:

RQ1: What is the associated overhead for mining process coverage?

RQ2: Is the associated overhead for mining process coverage less than for instrumentation-based coverage calculation?

For this we define a two factor/two treatments with-in group experiment design: The first independent variable is the coverage method (Instrumentation vs. Mining) and the second is the test suite size. Our dependent variable is the execution time of the measured test suites.

As subjects we used 4 BPEL processes, for which we could automatically – and thus unbiased – generated test suites of different sizes by using facet classification trees [16]. Two processes are taken from Schnelle [17] and two processes are taken from Terravis, which is an industrial project, which develops and runs a process-integration platform between land registers, notaries, banks and other parties across whole Switzerland [18].

### B. Data Collection

For running the experiment we set up a process engine on a dedicated virtual server together with the required infrastructure, e.g., the tools for measuring test coverage.

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-614-9 

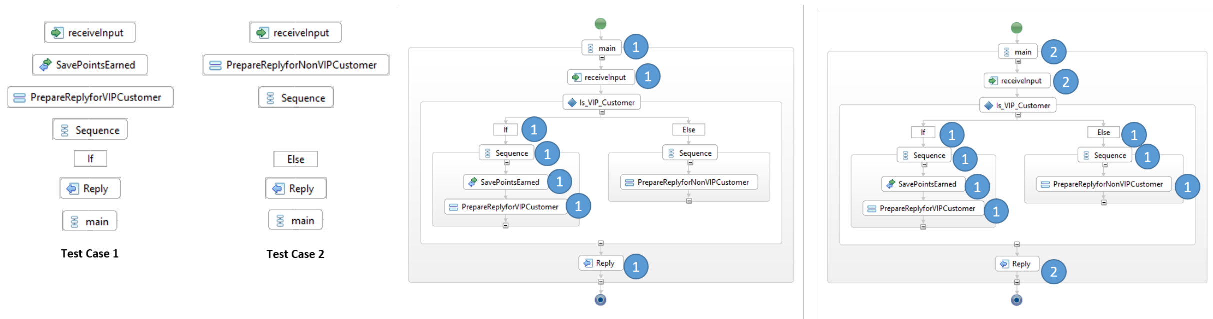


Figure 3. Event Logs for two Test Cases (Left), first Test Case replayed on BPEL Model (center) and both Test Cases replayed on BPEL Model (right).

| | Pre-Test | | Test | Post-Test | | |
|---|---|---|---|---|---|---|
| Instrumentation | Analyze Process Model | Instrument Process Model | Execute Test<br>Receive Markers | Replay Markers | | |
| Mining | | | Execute Test | Analyze Process Model | Fetch Event Log | Replay Events |

Figure 4. Comparison of Instrumentation and Mining.

Because the original BPELUnit tool for measuring test coverage [11] did not support vendor extensions and the deployment artefacts of the used process engine, we needed to re-implement the instrumentation tool with full support for all features, which are used by the industry project.

We measured the execution times by following the described process:

1) For every BPEL process, generate the test suites of different sizes,
2) For every test suite and for every calculation method, run 10x:
   a) Instrument the deployment unit (if necessary)
   b) Deploy the process,
   c) Run test suite,
   d) Wait for process log and calculate coverage (if necessary)

For every process, we generated random test suites with the sizes $n \in \{1, 5, 10, 25, 50, 75, 100\}$ if possible. The processes by Schnelle had only a smaller number of possible test cases, thereby the experiment could only use test cases with max. 25 and respectively 50 test cases.

We executed all test suites ten times in order to build mean values for all time measurements. All in all, 460 test suites runs were made for each coverage measurement method.

For all our test executions we used a virtual machine with 2 virtual CPUs and 4 GiByte of RAM running on Kubuntu with Informatica ActiveVOS 9.2 and MySQL.

### C. Results

The mean execution times of our measurements (calculated in milliseconds) are shown in Table **??**. T or S indicate the process set (Terravis or Schnelle), 1 or 2 indicate which process, and I or L indicate the coverage measurement method (instrumentation or log analysis).

The mean value for the smallest test suites with only one test case are smaller for instrumentation than for log analysis. For all other chosen test suite sizes, log analysis performs faster.

TABLE I. TOTAL MEAN EXECUTION TIME (ms)

| #TC | T1-I | T1-L | T2-I | T2-L | S1-I | S1-L | S2-I | S2-L |
|---|---|---|---|---|---|---|---|---|
| 1 | 8532 | 7718 | 5138 | 4915 | 4544 | 3825 | 4029 | 4140 |
| 5 | 16240 | 10958 | 11736 | 6533 | 6646 | 4492 | 6553 | 4249 |
| 10 | 19523 | 12446 | 14356 | 7090 | 10567 | 5942 | 9737 | 5586 |
| 25 | 38262 | 19309 | 34086 | 11760 | 15856 | 6688 | 18290 | 7675 |
| 50 | 62288 | 27264 | 62589 | 17736 | 29584 | 9799 | - | - |
| 100 | 120720 | 48628 | 118413 | 28318 | - | - | - | - |

By subtracting the normal execution time of a test suite we derive the absolute overhead (calculated in ms) as shown in Table II. In general, the numbers for log analysis are much lower than for instrumentation and do not increase that much. The highest overhead for log analysis is 6519ms in contrast for up to 94287ms for instrumentation. The overhead is the largest for the first Terravis process (T1) for process log analysis while it is the largest for instrumentation with the second Terravis process (T2).

TABLE II. ABSOLUTE OVERHEAD OF COVERAGE CALCULATION

| #TC | T1-I | T1-L | T2-I | T2-L | S1-I | S1-L | S2-I | S2-L |
|---|---|---|---|---|---|---|---|---|
| 1 | 2565 | 1751 | 2054 | 1830 | 1872 | 1153 | 1549 | 1660 |
| 5 | 7288 | 2006 | 7124 | 1920 | 3562 | 1408 | 3562 | 1258 |
| 10 | 9447 | 2370 | 9100 | 1834 | 6722 | 2098 | 5792 | 1641 |
| 25 | 22580 | 3628 | 24808 | 2482 | 10741 | 1574 | 12672 | 2057 |
| 50 | 39707 | 4683 | 48003 | 3150 | 21637 | 1852 | - | - |
| 100 | 78611 | 6519 | 94287 | 4193 | - | - | - | - |

We calculated the relative overhead for the processes by dividing the absolute overhead by the normal test suite execution time as shown in Table III. While for larger test suites the relative overhead increases with instrumentation, it decreases for log analysis. Relative overhead of instrumentation ranges between 43% and 391%, while it ranges bwetween 16% and 68% for log analysis.

The measurements grouped by coverage calculation method and process for all test suite runs are shown in Fig-

TABLE III. RELATIVE OVERHEAD OF COVERAGE CALCULATION

| #TC | T1-I | T1-L | T2-I | T2-L | S1-I | S1-L | S2-I | S2-L |
|---|---|---|---|---|---|---|---|---|
| 1 | 0.43 | 0.30 | 0.67 | 0.59 | 0.71 | 0.43 | 0.64 | 0.68 |
| 5 | 0.78 | 0.23 | 1.55 | 0.42 | 1.16 | 0.46 | 1.19 | 0.42 |
| 10 | 0.94 | 0.24 | 1.73 | 0.35 | 1.75 | 0.55 | 1.48 | 0.41 |
| 25 | 1.44 | 0.23 | 2.68 | 0.27 | 2.10 | 0.31 | 2.26 | 0.37 |
| 50 | 1.76 | 0.21 | 3.29 | 0.22 | 2.73 | 0.23 | - | - |
| 100 | 1.93 | 0.16 | 3.91 | 0.17 | - | - | - | - |

ure 5. Test suites with more test cases expectedly take longer to execute. Log analysis is usually faster than instrumentation.

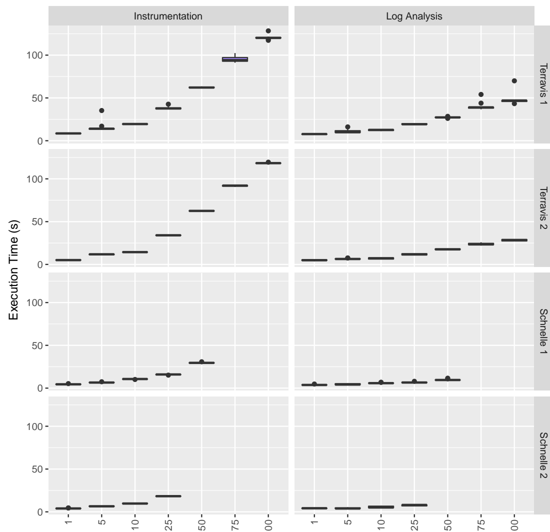


Figure 5. Overall Execution Times

The absolute and relative overhead of both coverage calculation methods are shown in Figure 6 and can be compared directly. Different colors indicate different processes. The absolute overhead shows clusters of overhead times that are associated with a test suite. As can be seen the values for both the absolute – and following from that – the relative overhead are higher most of the time for the instrumentation approach. Only in 13 of 460 measurements instrumentation was faster than log analysis. All of those measurements are concerned with test suites with only one test case.

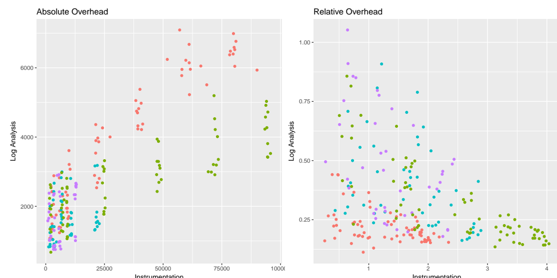


Figure 6. Coverage Measurement Overhead

In order to answer RQ2 we performed a paired, two-sided Wilcoxon hypothesis test with the null hypothesis $H_0$ being that no difference exists in the test suite execution times when using instrumentation or log analysis: Over all executed test suites, $p = 5.034 \times 10^{-12}$. However, we have seen that at least test suites with only one test case behaves differently from other test suites. Therefore, we blocked for the test suite size and derived the p-values as shown in Table IV. Values marked with (***) are less than 0.001 and thus highly statistical significant.

TABLE IV. P-VALUES (TWO-SIDED, PAIRED WILCOXON TEST) FOR TEST SUITES WITH n TEST CASES AND FOR ALL TEST SUITES

| Test Suite Size | p-Value |
|---|---|
| 1 | $0.087\,69$ |
| 5 | $1.25 \times 10^{-6}$ (***) |
| 10 | $5.154 \times 10^{-10}$ (***) |
| 25 | $2.918 \times 10^{-10}$ (***) |
| 50 | $1.691 \times 10^{-17}$ (***) |
| 75 | $1.451 \times 10^{-11}$ (***) |
| 100 | $1.451 \times 10^{-11}$ (***) |
| All | $5.034 \times 10^{-12}$ (***) |

### D. Interpretation

*1) RQ1: Overhead of Log Analysis:* Our measurements for the overhead of log analysisshow demonstrate that the absolute overhead increases and the relative overhead decreases with more test cases. The maximum absolute overhead of 6.5s for 100 test cases the performance penalty is little. This means that measuring approx. 92 test suites of such size would only impose a ten minute overhead (e.g., during nightly builds).

*2) RQ2: Overhead of Log Analysis compared to Instrumentation:* Our measurements clearly show that log analysis is significantly faster than instrumentation for non-trivial test suites, i.e. test suites with more than one test case. While the relative overhead of instrumentation increases with more test cases and reaches 391% (i.e., nearly quintuples the test suite execution time), log analysis imposes 68% overhead in the worst case of a small test suite but decreases to 16% for large test suites. For a further interpretation typical test case sizes in industry are needed in order to evaluate typical overhead ranges. If we suppose that a typical test suite consists of 25 test cases the relative overhead is between 24% and 55% for log analysis while it already is between 144% and 268% for instrumentation. This means that for any non-trivial test suite, log analysis brings a huge performance benefit when measuring test coverage.

### E. Threats to Validity

As with every empirical research there are associated threats to validity. Because we could only use four BPEL processes for our experiment, the question of generalizability arises.

Since we research technical effects only, the findings should be generalizable to all BPEL processes that execute a minimum threshold number of activities or test cases. The p-value for rejecting the null hypothesis and accepting that log analysis is faster than instrumentation for a test suite size $t \geq 5$ is so low that we are confident that replications will find the same results.

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-614-9 

As long as the process engine stores all relevant events that are required for calculating the test coverage metrics, the log analysis can proceed. To our knowledge, all BPEL engines are able to write event logs that contain the required event types. For every newly supported BPEL engine, a new interpreter of these events needs to be developed. The analysis and replay components can be reused. However, as part of our study we also found that this is also true for instrumentation tools: While BPEL is standardized, its extensions and the deployment artefacts are not.

The presented numbers are only applicable to automated unit tests. While we think it is safe to generalize the absolute overhead to other test scenarios, we expect that the relative numbers to be different: Manual tests take longer for executing the same number of processes, because user interactions take time, which makes the process duration longer. Thus, we do not think that the relative overhead can be generalized to other test types.

## VI. Conclusion & Future Work

Within this paper we presented a new approach to mine process event logs – which are usually already written when using a process execution engine – to calculate test coverage metrics of BPEL processes. Our new approach shows clear performance advantages over the instrumentation approach. Furthermore, the process mining approach can be used in other scenarios than the instrumentation approach: Because all activities for mining the test coverage are performed after the tests are run, it does not matter how the tests are run and when they were run. In contrast, coverage calculation needs a marker collection service running the whole time, which in practice is mostly only feasible during unit tests. Mining the process logs is completely independent of any test automation and can be used for automatic unit tests, automatic integration tests but also manual integration and system tests. The only drawback is, however, that the Process Engine needs to be configured to write the event log for all measured processes.

Although we have implemented test coverage mining for BPEL processes, the approach can be applied to other executable process languages as well: Process engine architectures are the same, e.g., BPMN 2.0 as the successor to BPEL defines other activities and is completely graph based. However, process engines executing BPMN 2.0 are also logging events for executed activities which can be replayed on top of BPMN 2.0 process models. Writing the process mining algorithm should be even simpler, because BPMN 2.0 defines process-wide unique identifiers for activities that are hopefully contained in the event log making reverse-engineering of vendor-specific identifiers obsolete.

While we have finished our research implementation, we want to optimize the implementation further and contribute it into the BPELUnit test framework. We hope to find further industry BPEL processes to apply our approach to and have a larger data set for evaluating performance – especially the use of other process engines is interesting and see whether all necessary event data is generally available.

Being able to calculate test coverage for non-unit tests also allows further research into executable process test methods: For example, experiments on the influence of different system testing approaches on test coverage.

## References


[1] D. Lübke, Test and Analysis of Service-Oriented Systems. Springer, 2007, ch. Unit Testing BPEL Compositions, pp. 149–171.

[2] P. Piwowarski, M. Ohba, and J. Caruso, "Coverage measurement experience during function test," in Proceedings of the 15th International Conference on Software Engineering, ser. ICSE '93. Los Alamitos, CA, USA: IEEE Computer Society Press, 1993, pp. 287–301.

[3] J. R. Horgan, S. London, and M. R. Lyu, "Achieving software quality with testing coverage measures," Computer, vol. 27, no. 9, Sept 1994, pp. 60–69.

[4] L. C. Briand, Y. Labiche, and Y. Wang, "Using simulation to empirically investigate test coverage criteria based on statechart," in Proceedings. 26th International Conference on Software Engineering, May 2004, pp. 86–95.

[5] Y. K. Malaiya, M. N. Li, J. M. Bieman, and R. Karcich, "Software reliability growth with test coverage," IEEE Transactions on Reliability, vol. 51, no. 4, Dec 2002, pp. 420–426.

[6] X. Cai and M. R. Lyu, "Software reliability modeling with test coverage: Experimentation and measurement with a fault-tolerant software project," in The 18th IEEE International Symposium on Software Reliability (ISSRE '07), Nov 2007, pp. 17–26.

[7] D. Lübke, "Using Metric Time Lines for Identifying Architecture Shortcomings in Process Execution Architectures," in Software Architecture and Metrics (SAM), 2015 IEEE/ACM 2nd International Workshop on. IEEE, 2015, pp. 55–58.

[8] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," ACM Comput. Surv., vol. 29, no. 4, Dec. 1997, pp. 366–427. [Online]. Available: http://doi.acm.org/10.1145/267580.267590

[9] Z. Li, W. Sun, Z. B. Jiang, and X. Zhang, "BPEL4WS Unit Testing: Framework and Implementation," in ICWS '05: Proceedings of the IEEE International Conference on Web Services (ICWS'05). Washington, DC, USA: IEEE Computer Society, 2005, pp. 103–110.

[10] P. Mayer and D. Lübke, "Towards a BPEL unit testing framework," in TAV-WEB '06: Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications. New York, NY, USA: ACM Press, 2006, pp. 33–42.

[11] D. Lübke, L. Singer, and A. Salnikow, "Calculating BPEL Test Coverage through Instrumentation," in Workshop on Automated Software Testing (AST 2009), ICSE 2009, 2009, pp. 115–122.

[12] W. l. Dong, H. Yu, and Y. b. Zhang, "Testing bpel-based web service composition using high-level petri nets," in 2006 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC'06), Oct 2006, pp. 441–444.

[13] Informatica. Bpel unit testing. [Online]. Available: http://infocenter.activevos.com/infocenter/ActiveVOS/v92/index.jsp?topic=/com.activee.bpep.doc/html/UG21.html (2016)

[14] Oracle. Oracle bpel process manager developer's guide: Testing bpel processes. [Online]. Available: https://docs.oracle.com/cd/E11036_01/integrate.1013/b28981/testsuite.htm (2007)

[15] W. van der Aalst, Process Mining – Data Science in Action. Springer, 2016.

[16] T. Schnelle and D. Lübke, "Towards the generation of test cases for executable business processes from classification trees," in Proceedings of the 9th Central European Workshop on Services and their Composition (ZEUS) 2017, 2017, pp. 15–22.

[17] T. Schnelle, "Generierung von bpelunit-testsuites aus klassifikationsbäumen," Master's thesis, Leibniz Universität Hannover, Fachgebiet Software Engineering, 2016.

[18] W. Berli, D. Lübke, and W. Möckli, "Terravis – large scale business process integration between public and private partners," in Lecture Notes in Informatics (LNI), Proceedings INFORMATIK 2014, E. Plödereder, L. Grunske, E. Schneider, and D. Ull, Eds., vol. P-232, Gesellschaft für Informatik e.V. Gesellschaft für Informatik e.V., 2014, pp. 1075–1090.