An Extended Evaluation of Process Log Analysis for BPEL Test Coverage Calculation

Daniel Lübke

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

***Abstract*—With today's requirement of quickly developing digitization solutions, companies often use specialized workflow languages like the BPEL or BPMN 2.0, which orchestrate services along the process flow. Because process models are of critical importance to the functioning of the organization, high quality and reliability of the implementations are mandatory. Therefore, testing becomes an even more important activity in the development process. For judging the quality of developed tests, Test Coverage Metrics can be used. Current approaches to test coverage calculation for BPEL either rely on instrumentation, which is slow, or are limited to vendor-provided unit test frameworks, in which all dependent services are mocked, which limits the applicability of such approaches. Our refined approach relies on analyzing process event logs that are written during process execution. Within this article we analyze the performance characteristics of process log analysis versus the instrumentation-based approach by running an experiment with BPEL processes and their accompanying test suites developed in an industry project. According to our findings, the improved version of process log analysis is significantly faster for all scenarios.**

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

I. Introduction

This article presents improvements, an extended description as well as an improved experimental evaluation of using process log analysis as a method for measuring test coverage of BPEL processes presented earlier [1].

Having a flexible and fast tool for measuring test coverage is important due to the rising importance of digital business process solutions. Partner networks are being connected tighter and the integration between different businesses is often driven by business process needs. For example, offering fully fledged digital services to customers requires a high degree of automation, i.e., the implementation of large parts of business processes in software solutions. Because the failure of customer- or partner-facing processes can have dramatic financial and reputational consequences, the required quality, stability and correctness of process-based software solutions is an important problem in practice – and as such a relevant research topic.

Business processes can be digitized by using special workflows, which are refered to as executable business processes. Standards like BPEL or BPMN 2.0 have been developed to automate business processes in large companies by orchestrating services. These 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 perform good quality assurance in order to avoid costly problems in production [2]. Quality Assurance can include static checking of process models (e.g., consistency check of service contracts to executable processes [3]). However, most projects use testing as their main quality assurance activity. Consequently, they require an assessment of the adequacy and quality of the tests. It has been shown by Piwowarski et al. [4] 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. [5], who linked data-flow testing metrics to reliability, and Braind et al. [6], who simulated the impact of higher test coverage on quality. Furthermore, Malaiya et al. [7] and Cai & Lyu [8] have developed prediction models that can link test coverage to test effort and software reliability.

Quality Assurance, and thus test coverage measurement, should be an ongoing activity because executable processes will evolve over time [9]. One way for continously measuring test quality is to measure test coverage as part of all testing activities. Test Coverage then serves as measurement of test data adequacy [10].

While approaches applicable for developing unit tests for executable processes have been proposed by academia (e.g., [11], [12]) 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 unit 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 [13]. This is far more than instrumentation approaches for "normal" programming languages, e.g., Java, require.

An approach that better guides quality assurance in software projects, which develop executable processes, is required. This approach shall be applicable in several test scenarios, including unit tests, integration tests and system tests. Ideally, it is easier to set up than existing methods in order to improve acceptance by practitioners [4].

Within this article, we evaluate an approach based on analyzing process event logs, which are automatically written

by process engines during process execution regardless of whether testing frameworks are used or not. The evaluation is done experimentally and compares the execution times of two test coverage measurement approaches: process log analysis and process instrumentation. For running the experiment, four BPEL processes from the commerical Terravis project [14] as well as two research processes from Schnelle [15] are used.

This article is structured as follows: First, the process modeling language BPEL is shortly explained in Section II before related work is presented in Section III. The approach for mining test coverage metrics is described in detail in Section IV followed by a short evaluation of its flexibility in Section V. The main part of this article is presented in Section VI, which describes the experiment set-up, the gathered results and their interpretation comparing the performance of our new approach and the existing instrumentation-based approach. Finally, we conclude and give an outlook on possible 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 service others to call a process via service interfaces). Important *Structured Activities* are the *if*, *while*, *repeatUntil*, and *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 defines *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 within the process, 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 the BPMN 2.0 standard does, but standardizes the XML format, in which it is saved. Vendors have developed their own graphical representations. Within this article we use the notation of the Eclipse BPEL Designer. A process that will be used as an example in this article is shown in Figure 1: A customer places an order (“receiveInput”). 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”). For handling failures while storing the earned points, a *Fault Handler* called “catch” is defined, which prepares (“PrepareTicketCreation”) and creates a help desk ticket (“CreateHelpDeskTicket”) so that the points can be manually added later.

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 in most engines during the deployment of a process model.

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

Testing BPEL processes has become subject of many research projects. For example, Li et al. (BPEL4WS Unit Testing Framework [11]), Mayer & Lübke (BPELUnit [12]), and Dong et al. (Petri Net Approach to BPEL Testing [16]) have developed approaches for testing BPEL processes and published their ideas.

The BPELUnit framework was developed by Mayer & Lübke [12] and was later extended by Lübke et al. [13] with test coverage measurement support. First, the coverage metrics needed to be defined, which is not as straightforward as for other programming languages due to BPEL’s different mechanisms for defining the process-flow. Consequently, three coverage 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 execution position (“markers”) to the test framework. This enables the test framework to know which activities have been executed in the test run. However, the test framework needs to run while the instrumented processes are executed in order to collect the markers, which makes its use limited in practice. Even more, the overhead introduced by many new service calls is considerable: The reported overhead in the original paper is more than 100%, i.e., the test execution times have more than doubled. This stems from the instrumentation mechanism, which requires every execution trace point to be sent out of the process via a service call, which in turn 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 it only slightly depends on the Process Engine being used: The changes to the BPEL process are completely standards-compliant. Only the new service for collecting markers needs to be added to the engine-specific deployment descriptor. One way to migitate the performance problems is to distribute the tests, e.g., as described by Kapfhammer [17].

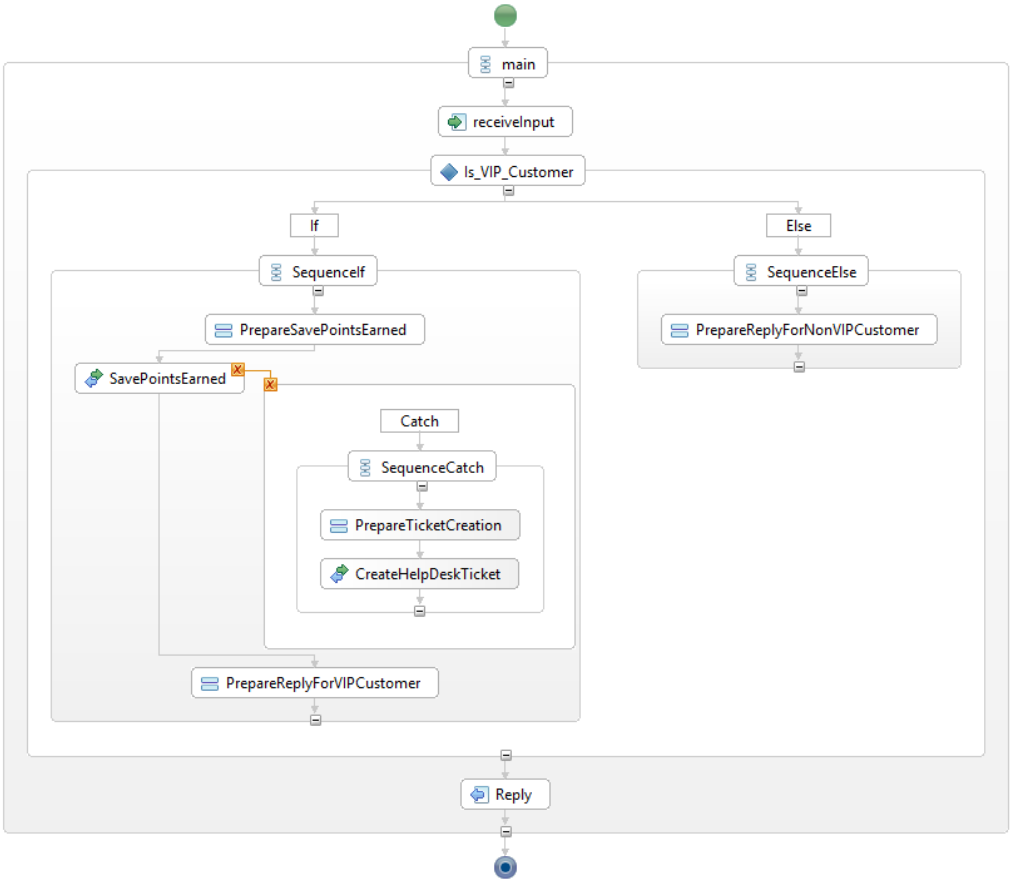

Figure 1. Sample BPEL Process for processing an Order.

Process Engine vendors have also developed their own proprietary solutions for measuring test coverage: 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 structures, its use is limited to unit test scenarios only. Examples of such vendor-provided test frameworks are Informatica's BUnit [18] and Oracle's BPELTest [19].

Endo et al. [20] defined coverage criteria for their test generation approach. For example, one criterion called All-nodes is that all activities are executed. Other criteria require certain activity types to be executed (e.g., All-nodes for all *invoke* and *reply* activities). The authors use these criteria to guide the test case selection of their generation approach.

On a more general level, test metrics and their publication by services themselves have been researched in the context of Service-Oriented Architectures (SOA) by Miranda et al. [21], Bartolini et al. [22], and Eler et al. [23]. Their approaches are independent of the language used for implementing the services and thus more abstract.

Schnelle & Lübke proposed an approach to generate unit test cases from classification trees, which are designed from a business perspective [24]. Coverage can be specified as the coverage of different properties (=leafs) of that tree. The coverage is not code-based but instead requirements-based.

Other approaches try to generate test cases with good or optimal coverage: Kaschner & Lohmann [25] developed an approach that generates test suites that cover all service interactions. Service Interactions are externally observable behavior and are thus deemed the most important aspect to test by the authors. Ji et al. [26] describe another way: They developed an algorithm that tries to choose the most efficient test cases by analyzing the data-flow of a BPEL process.

Although many coverage metrics have been defined, Weiser et al. [27] have shown that "[e]mpirically comparing structural test coverage metrics reveals that test sets that satisfy one metric are likely to satisfy another metric as well". This means that for practical purposes any of the proposed test coverage metrics will likely behave as well or as badly as another one.

All test approaches available for BPEL claim that they achieve – or at least help to achieve – a good test coverage.

However, no empirical studies have been made whether this is the case and what typical test coverage values constitute for BPEL processes. In contrast, there is a huge body of knowledge available for general purpose programming languages, as has been shown in a survey for the Java programming language by Yang et al. [28].

In a multiple case study approach Mockus et al. [29] have shown that with increasing test coverage more defects are found prior to production but that costs to improve test coverage increase exponentially while the numbers of found defects only increases linearily. Therefore, Pavlopoulou & Young [30] propose to monitor the production environment for execution of previously untested code sections. By leaving only instrumentation code for these code areas of interest, they could reduce the overhead imposed by coverage measurements.

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 presents the different steps of our approach that are performed for analyzing process logs in order to calculate test coverage.

A. Metric Calculation Process

For calculating test coverage, we use process mining techniques. Process Mining is concerned with building "a strong relation between a process model and 'reality'captured in the form of an event log" [31, p. 41]. Although process mining techniques are usually used to help the business (e.g., [32], [33], [34]), it is used here to guide the development project: By having the BPEL process model and the event logs of all test cases available from the process engine's database, 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 [31, p. 43].

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

1) Build the BPEL Process Model Syntax Tree from its XML representation (BPEL Analysis),
2) Wait until the whole event log has been written,
3) Fetch the event log from the Process Engine (Data Gathering),
4) 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 [13] and thus takes the same time to build. All BPEL Models are accessible via the process engine's repository and can be extracted as part of the coverage mining. This guarantees that the event logs match the process model versions exactly.

C. Wait for the Event Log

Unfortunately, the used BPEL engine writes the event log asynchronously and delayed during process execution. This means that there is a delay between process completion and the event log being written to the database. The persistence interval can be configured. In the initial version of process log analysis for test coverage calculation [1], a fixed delay before reading the event logs was introduced that was a long as the configured persistence interval. This configuration specific delay is a fixed cost penality before test coverage calculation starts. The improved version of our implementation actually queries the event log as long as the end event for the last process instance has been written. This should reduce the wait time on average by half.

D. Data Gathering

The BPEL processes of the industry project, which we use in our experiment, uses Informatica ActiveVOS [18] as its BPMS. ActiveVOS is a process engine fully compliant with the BPEL 2.0 and BPEL4People standards and stores all data – especially all available process models in all versions, active and completed process instances, and event logs – in a relational database. This allows access to and analysis of the available data that can be mined for calculating 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* and the same for loop and branching conditions (*condition evaluated to true* and *condition evaluated to false*). Besides the event type, the event timestamp, the corresponding process instance, and an internal activity or link identifier is logged.

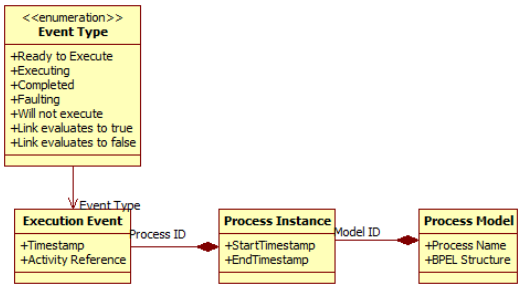

Figure 2. Conceptual Data Model of the Process Engine being used.

This means that all necessary data is available for reconstructing the execution of a process instance and thereby calculating the test coverage metrics: For calculating activity and handler coverage, all *completed*, and *faulting* events need to be fetched for a given test run. For calculating link coverage both *link evaluated* events need to be fetched and for calculating

branch coverage, both *condition evaluated* events need to be fetched as well.

All other event types – especially *Will not execute* events 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.

E. Replay & Metric Calculation

Test Coverage Metrics – as defined by Lübke et al. [13] – is calculated with the data extracted in the previous steps. At first, all activities, handlers 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 and handlers that have a corresponding *completed* or *faulting* event are marked as being executed. Also, conditional BPEL links for graph-based modeling are marked with the link evaluation events. However, every link can carry two different markers: one if the condition was evaluated to true and another if the condition was evaluated to false. Because links without a condition are excluded from the coverage metric, they are ignored from further analysis.

Similarily, loops are being marked according to the *condition evaluated* events. During this phase, loop activities can be marked as executed twice for calculating the branch coverage in later stages. For *if* or *ifElse* branches and for most loops markers can be set, whenever the condition is evaluated to true; except for the *repeatUntil*, which follows an inverted boolean logic, and the markers are set if the condition is evaluated to false. The only exception is the parallel *forEach* loop, in which the activity identifier contains the number of the currently executed parallel branch. If a counter larger than one is encountered, the *forEach* activity is marked as executed at least twice.

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

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

This metric can be filtered by basic activity type. For example, the coverage of all executed *invoke* activities can be calculated as shown in equation (2).

$$C_{invoke} := \frac{|A_m^{Invoke}|}{|A_m^{Invoke}| + |A_u^{Invoke}|} \tag{2}$$

Similarily, Handler Coverage C_H can be calculated by searching the syntax tree for handlers that have been successfully executed as shown in equation (3): The coverage is the proportion of executed handlers in relation to all handlers.

$$C_H := \frac{|H_m|}{|H_m| + |H_u|} \tag{3}$$

With the given Process Engine it was important to not mark event handlers when they are *ready to execute* because this event will be triggered by the Process Engine whenever the context of the handler gets activated and the handler might be triggered and not when the handler really starts executing or is completed. We use the completion events of the first contained basic activity in an event handler. A handler has been executed, if and only if its first basic activity has been completed successfully or unsucessfully.

This metric can again be filtered for different handler types, e.g., fault handlers, as shown in equation (4).

$$C_{faultHandler} := \frac{|H_m^{fault}|}{|H_m^{fault}| + |H_u^{fault}|} \tag{4}$$

Link Coverage C_L as defined in equation (5) determines what fraction of conditional links in *flow* activities has been evaluated to *true* and *false* respectively.

$$C_L := \frac{|L_m^+| + |L_m^-|}{|L_m^+| + |L_m^-| + |L_u^+| + |L_u^-|} \tag{5}$$

The ActiveVOS BPEL engine logs *transition condition evaluated events*, which also contain the evaluation results. This is very different compared to the instrumentation approach, which requires heavy model modifications in order to distinguish between links that have been subject to BPEL's dead path elimination [35] or which have been really evaluated to false. Because of this, link coverage can be easily calculated by traversing the marked syntax tree. The set L is the set of all conditional links, L_m^+ are all conditional links that have been marked as being executed with the condition evaluated to true, L_m^- are all conditional links that have been marked as being executed with the condition being evaluated to false, L_u^+ are all conditional links that are not marked as being executed with the condition being true, and L_u^- are all conditional links that are not marked as being executed with the condition being false.

Branch Coverage C_B metric complements Link Coverage: Branch Coverage includes all edges in the control-flow graph of structured BPEL activities, i.e., *if*, parallel and sequential *forEach*, *while*, and *repeatUntil* activities but does not include the links in the graph-based *flow* activity, which are only covered by link coverage as defined above. The main problem is that this metric needs to count executions of edges that are not necessarily part of the BPEL model: an *if* does not need to have an *else* and the loops have no edges returning to the loop start and can even support parallel execution like the parallel *forEach* loop. Thus, no completion events can be used but instead other events or further analysis of the model are required. In case of the ActiveVOS BPEL engine, all sequential loops are handeled by using *condition evaluation* events similar to the calculation of link coverage. Parallel *forEach* loops can be measured by parsing the activity identifier in the event, which contains an instance number: If the instance counter is larger than 1 the *forEach* "loops" more than once.

Branch Coverage can be calculated according to Equation (6): The number of *forEach* (F), *repeatUntil* (R) and *while* (W) activities, which have been marked as not executed,

executed once or executed more than once are divided by all possible markers, which are three markers for the *forEach* and *while* activities, which can have arbitrary loop counts, and two markers for the *repeatUntil* activitiy, which must be looped at least once.

$$C_L := \frac{|F_m^{0,1,*}| + |W_m^{0,1,*}||R_m^{1,*}|}{3 \cdot |F_a| + 3 \cdot |W_a| + 2 \cdot |R_a|} \quad (6)$$

Depending on the BPEL modeler's choice between using BPEL's *flow* activity or the block-structured activities, link or branch coverage is more meaningful.

When conducting another research project, we accidentially found that – in contrast to other programming languages – the branch coverage is not stricter than the activity coverage. If an event handler – and consequently its basic child activities – is not executed, branch coverage can be complete but the basic activity coverage is not. The problems arises due to the original split of control-flow related coverage metrics into branch, link and handler coverage.

Therefore, we define the new coverage metric Conditional Coverage for BPEL processes that unifies all conditional control-flow metrics. The measurement of all values follows the rules as described for the other coverage metrics above.

F. Example

To illustrate the replay of the event log on top of the process model we assume three test cases for the example BPEL process as shown in Figure 3. The first test case tests the VIP Customer, the second one the Non-VIP Customer and the third the VIP Customer with a problem when booking bonus points.

The "completed" events generated by the Process Engine for the first test case are

1) Receiving the start message (*receive* activity "ReceiveInput"),
2) creating a message for storing the points(*assign* activity "PrepareSavePointsEarned"),
3) calling a service to credit points (*invoke* activity "SavePointsEarned"),
4) creating the response message (*assign* activity "PrepareReplyforVIPCustomer"),
5) completing the sequence within the *if* (*sequence* activity "Sequence"),
6) completing the *if* (*if* branch "If and *if* activity "Is_VIP_Customer"),
7) sending the reply (*reply* activity "Reply"), and
8) finally completing the main sequence (*sequence* activity "main").

As can be seen in the traces in Figure 3, 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 left of Figure 4. Additionally replaying the second and third test case yields the markings as shown on the right hand side of the same figure. The numbers in the markers denote how often the activity has been executed. With these three test cases, all basic activities are covered, i.e., all basic activities have been executed at least once, all branches are covered, i.e., both the "if" and "else" branch have been executed, and all handlers are covered, i.e., the "catch" handler is executed at least once.

V. Comparison to Instrumentation

When we compare our approach to instrumentation (see Figure 5), 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 change the BPEL process model while process mining does not,
- as a consequence 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, 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 conceptual differences, our approach is more flexible than instrumentation because it defers the decision whether to calculate test coverage on a given test run: it is possible that coverage is calculated without preparation after testing has completed and if the event logs are still available.

We expect our log analysis approach also 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 the execution of the executable processes 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. This hypothesis is tested in an experiment described in the next section.

VI. Experiment

In order to evaluate the presented approach, we conducted an experiment that is described in this section.

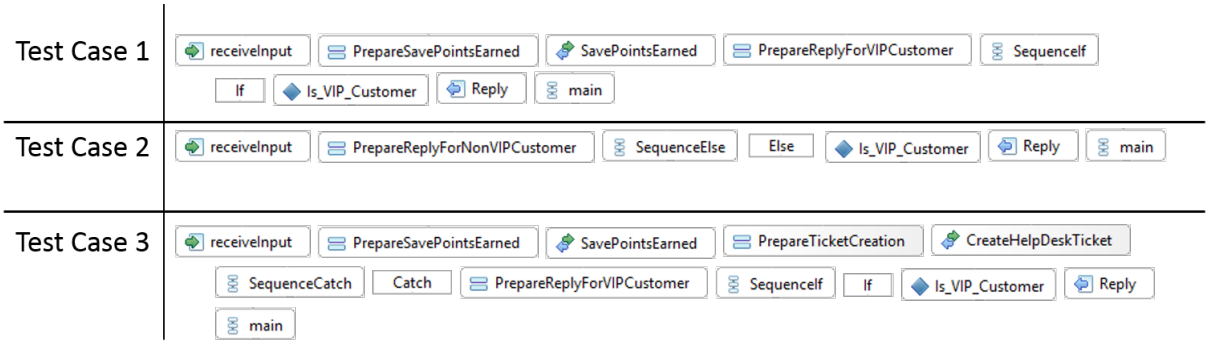

Figure 3. Event Traces for Different Test Cases.

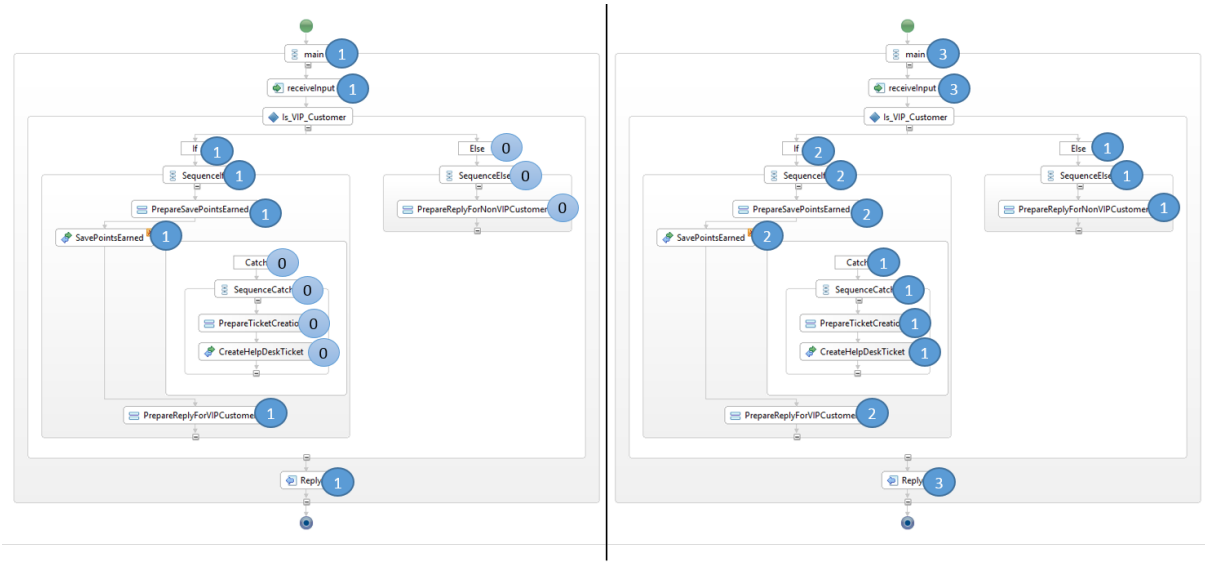

Figure 4. Markers for First Test Case (left) and all Test Cases (right).

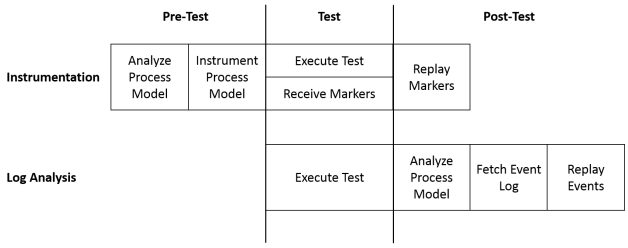

Figure 5. Comparison of Instrumentation and Mining.

A. Experiment Description & Design

For evaluating the performance implications of our approach, we conduct an experiment, in which we want to answer the following research questions:

- RQ1: What is the associated overhead for instrumentation-based coverage calculation?
- RQ2: What is the associated overhead for mining process coverage?
- RQ3: When is the associated overhead for mining process coverage less than for instrumentation-based coverage calculation?
- RQ4: Does the size of the test suite influence the overhead of mining coverage calculation?
- RQ5: Does the size of the test suite influence the overhead of instrumentation-based coverage calculation?

In order to find answers to these questions we define a two factor/two treatments with-in group experiment design: The first independent variable is the coverage method (Instrumentation vs. Log Analysis) and the second is the test suite size. The dependent variable is the execution time of the measured test suites.

As subjects we used 6 BPEL processes, for which tests based on classification trees are available [24]. Classification Trees allow for a generator-based approach for creating test suites. By randomly selecting a subset of test cases, test suites of configurable sizes can be generated. This yields the advantage that test suites of arbitrary sizes can be generated, so that the test suite size can be controlled in the experiment and each process can be tested with different test suites. Four processes have been developed within the industry project

Terravis, which is an industrial project that develops and runs a process-integration platform between land registers, notaries, banks and other parties across whole Switzerland [14]. Two additional processes, which had classification trees for automated testing, are taken from Schnelle [15]. Latter were originally developed using the Eclipse BPEL Designer and Apache ODE but the vendor-specific configuration was added for ActiveVOS as part of this experiment in order to use the same environment including process engine and test coverage tools.

Process descriptions of all processes that were subjects in this experiment are are shown according to the process classification proposed by Lübke et al. [36] in Table I.

B. Data Collection

1) Environment and Measurement Process: 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.

The experiment was conducted by executing the following steps for every test suite:

1) Reset database and start BPMS,
2) Instrument the deployment unit,
3) Deploy the instrumented deployment unit,
4) Run the test suite with the marker collector,
5) Deploy the original deployment unit,
6) Run the test suite,
7) Wait for process log and calculate coverage,
8) Shutdown BPMS.

We chose to alternate the deployments of the instrumented and non-instrumented process versions in order to not allow the BPMS to optimize the deployment by reusing the old process definitions.

For every process, we generated random test suites with the sizes $n \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 25, 50, 75, 100\}$ if possible. Some processes had only a smaller number of possible test cases, thereby the experiment could only use test suites with max. 25 and respectively 50 test cases for these processes.

We repeated these test runs 20 times in order to build representative mean values for all time measurements. All in all, 2920 test suite runs were made.

We used a virtual machine with 2 virtual CPUs and 4 GiByte of RAM running on Kubuntu with Informatica ActiveVOS 9.2 and MySQL for all our test executions. The search indexing of both the host and virtual machine operating system were disabled in order to not have load unrelated to the experiment and the computer was taken offline in order to further shield it from unexpected load.

ActiveVOS was configured with all necessary settings for executing all processes. This especially includes custom project-specific extensions and service-wide configured settings. The configuration also specified a 1 second write delay for storing the event log.

2) Sample Implementation: We implemented a tool that performs the previously defined 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 required events. The coverage measurement tool, which uses log analysis, is available as open source[1].

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, post-processing of the event log data is required in order to properly resolve the referenced activities.

One additional problem we encountered while developing the sample implementation was BPEL's lack of unique identifiers for activities: The activity labels are not necessarily unique and can be defined by the designer without uniqueness constraints. Due to this, BPMS vendors are forced to build their own ways of identifying activities. We had to reverse-engineer the way the used process engine creates identifiers in the process log. Internally, our implementation uses XPath expressions that evaluate to a single activity by either matching a unique name – if one exists – or the position of the activity in BPEL's XML tree. We wrote a mapper, which rewrites the event log's activity identifiers to valid XPath expressions. This step is highly specific to the process engine being used and requires reverse-engineering the format and construction rules for the proprietary identifiers.

In contrast to our first experiment published previously [1], which used a fixed waiting time, we improved the waiting process by reading the highest process identifier from the database and see whether a process completion or failing event was written for that process instance. Because no other events can follow, this means that the whole event log is available. In order to guard against test cases that fail to complete a process, we added a maximum wait time that equals the write delay of the process log.

However, we also needed to re-implement the instrumentation tool: Because the original BPELUnit tool for measuring test coverage [13] did neither support vendor extensions nor the deployment artefacts of the used process engine, we needed to re-implement the instrumentation tool with full support for these features, which are used by the industry project.

C. Results

The mean execution times of our measurements (calculated in milliseconds) are shown in Table II. T or S indicate the process set (Terravis or Schnelle), 1 to 4 indicate which process from this set, and N, I or L indicate normal execution (N) or the coverage measurement method (I for instrumentation and L for log analysis.)

For all other chosen test suite sizes, log analysis performs faster than instrumentation.

By subtracting the normal execution time of a test suite we derive the absolute overhead (calculated in ms) as shown in Table III. 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 4513ms in contrast

[1] http://www.daniel-luebke.de/net.bpelunit.tools.coveragecalculator.zip

TABLE I. PROCESS CLASSIFICATION OF SUBJECTS IN THIS EXPERIMENT

	Online Shop (S1)	Credit Approval (S2)	Land Register Notifications (T1)	Depot Check (T2)	Transfer Approval (T3)	Register of Commerce (T4)
Version	-	-	-	-	-	-
Domain	E-Commerce	Banking	Mortgage Transactions			Register of Commerce
Geography	None	None	Switzerland			
Time	2016	2016	2018			
Boundaries	-	-	Cross-Organizational	Within-Dep.	Cross-Organizational	
Relationship	No call	No call	Is being called	No Call	Calls another	Calls another/ Is begin called
Scope	Core	Core	Auxilliary	Auxilliary	Core	Core
Purpose	Execution					
People Involvement	None	None	None	None	None	None
Process Language	BPEL 2.0	BPEL 2.0	BPEL 2.0 plus vendor extensions			
Execution Engine	Apache ODE		Informatica ActiveVOS 9.2			
Model Maturity	Illustrative	Illustrative	Productive			
Basic Activities	19	25	84	46	33	39
Structured Activities	8	12	89	46	34	33
Non-linear Struct.A.	6	14	46	20	9	4

TABLE II. TOTAL MEAN EXECUTION TIME (ms)

#TC	S1-N	S1-I	S1-L	S2-N	S2-I	S2-L	T1-N	T1-I	T1-L	T2-N	T2-I	T2-L	T3-N	T3-I	T3-L	T4-N	T4-I	T4-L
1	1973	3745	2613	1933	3589	2600	4534	7704	5231	2600	5003	3329	3888	6506	4615	4543	8019	5328
2	2244	4958	2941	2181	4473	2808	5190	10123	5927	2975	6675	3731	5421	9759	6196	4825	8535	5533
3	2287	5078	2919	2319	5516	3055	5162	10032	5950	3057	7405	3871	6606	11792	7440	5134	10482	5820
4	2409	5503	3111	2405	5831	3091	5733	11767	6547	3537	10761	4307	8004	14710	8743	5570	13540	6366
5	2471	5645	3107	2389	5890	3050	5499	11863	6427	4035	12886	4968	9405	17883	10132	5562	12774	6317
6	2732	7017	3480	2510	6486	3174	6050	13996	7012	3755	11545	4577	10721	20697	11538	5901	15428	6716
7	2853	7567	3570	3150	10248	3896	6524	15432	7455	4220	14224	5118	12040	23447	12866	5853	15131	6709
8	3156	9411	3795	2706	7668	3373	6827	16618	7828	4288	14442	5176	12967	24095	13740	6452	18699	7294
9	3142	9039	3887	3158	10429	3973	6801	16765	7838	4737	18490	5708	14951	29809	15826	6731	20277	7522
10	3250	9673	3989	3010	9692	3742	7576	18652	8656	5091	19469	6211	15859	30656	16729	7097	21889	7928
25	5164	19569	5914	4725	17369	5522	11187	34880	12874	8060	41445	9368	35033	70271	36183	10623	44380	11731
50	-	-	-	7193	36245	8113	17349	62530	19976	11727	73215	13571	-	-	-	-	-	-
75	-	-	-	7623	37552	8571	23502	90296	27161	15244	105404	17536	-	-	-	-	-	-
100	-	-	-	-	-	-	29746	117730	34259	18966	141603	21926	-	-	-	-	-	-

for up to 122637ms for instrumentation. The overhead is the largest for the second Terravis process (T2) for process log analysis while it is the largest for instrumentation with the first Terravis process (T1).

We calculated the relative overhead for the processes by dividing the absolute overhead by the normal test suite execution time as shown in Table IV. While for larger test suites the relative overhead increases with instrumentation, it decreases for log analysis. Relative overhead of instrumentation ranges between 67.3% and 647.3%, while it ranges bwetween 3.3% and 34.3% for log analysis.

The measurements grouped by coverage calculation method and process for all test suite runs are shown in Figure 6 side by side for comparison: Test suites with more test cases expectedly take longer to execute and log analysis is always faster than instrumentation.

The absolute and relative overhead of both coverage calculation methods are shown in Figure 7. Different colors in both charts 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 is always higher for the instrumentation approach.

In order to answer RQ3 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.

We calculcated the effect size as the absolute difference in execution time between both methods as well as the p-value for all combinations of test suite size and process (see Table V).

In the next step we analyzed the overhead of test coverage calculation in relation to the whole test suite size in terms of test activities, i.e., the activities in a test suite across all test cases.

Figure 8 shows the relationship between number of test activities and test execution time. The relationship is nearly perfectly linear: The more test activities are executed as part of a test suite the longer test execution takes. However, the slope of the linear relationship depends on the process under test.

In the next step we analyzed the relationship between the number of test activities and the absolute overhead of coverage calculation as shown in Figure 9 (please note that the y-axis scale is different for instrumentation and log analysis.) Both instrumentation and log analysis have a nearly perfectly linear increase of test duration. However, log analysis has a much

TABLE III. ABSOLUTE MEAN OVERHEAD OF TEST COVERAGE CALCULATION (ms)

#TC	S1-I	S1-L	S2-I	S2-L	T1-I	T1-L	T2-I	T2-L	T3-I	T3-L	T4-I	T4-L
1	1772	640	1656	666	3170	697	2403	728	727	2617	785	3475
2	2714	697	2291	626	4933	737	3700	757	776	4338	708	3710
3	2791	632	3197	736	4870	788	4348	813	834	5186	686	5348
4	3094	702	3427	687	6034	814	7223	769	739	6705	796	7970
5	3174	636	3502	661	6364	928	8851	933	727	8478	755	7211
6	4284	747	3976	664	7945	962	7790	822	817	9976	815	9527
7	4715	717	7098	746	8908	931	10004	898	827	11408	856	9278
8	6255	639	4962	667	9792	1002	10154	888	773	11128	842	12248
9	5898	746	7272	815	9964	1037	13753	971	875	14857	792	13546
10	6424	739	6681	732	11076	1080	14378	1120	870	14797	831	14792
25	14406	750	12644	797	23693	1687	33385	1309	1151	35238	1108	33757
50	-	-	29052	921	45182	2627	61488	1844	-	-	-	-
75	-	-	29929	949	66793	3658	90161	2292	-	-	-	-
100	-	-	-	-	87985	4513	122637	2960	-	-	-	-

TABLE IV. RELATIVE MEAN OVERHEAD OF TEST COVERAGE CALCULATION

#TC	S1-I	S1-L	S2-I	S2-L	T1-I	T1-L	T2-I	T2-L	T3-I	T3-L	T4-I	T4-L
1	0.90	0.32	0.86	0.34	0.70	0.15	0.93	0.28	0.19	0.67	0.17	0.77
2	1.21	0.31	1.06	0.29	0.95	0.14	1.25	0.25	0.14	0.80	0.15	0.77
3	1.23	0.28	1.40	0.32	0.95	0.15	1.42	0.27	0.13	0.79	0.13	1.04
4	1.29	0.29	1.43	0.29	1.05	0.14	2.04	0.22	0.09	0.84	0.14	1.43
5	1.29	0.26	1.47	0.28	1.16	0.17	2.20	0.23	0.08	0.90	0.14	1.30
6	1.57	0.27	1.59	0.27	1.31	0.16	2.08	0.22	0.08	0.93	0.14	1.62
7	1.66	0.25	2.26	0.24	1.37	0.14	2.37	0.21	0.07	0.95	0.15	1.59
8	1.99	0.20	1.84	0.25	1.44	0.15	2.37	0.21	0.06	0.86	0.13	1.90
9	1.88	0.24	2.31	0.26	1.47	0.15	2.91	0.21	0.06	0.99	0.12	2.02
10	1.98	0.23	2.22	0.24	1.46	0.14	2.83	0.22	0.05	0.93	0.12	2.09
25	2.79	0.15	2.71	0.17	2.12	0.15	4.15	0.16	0.03	1.01	0.10	3.19
50	-	-	4.06	0.13	2.61	0.15	5.25	0.16	-	-	-	-
75	-	-	3.92	0.13	2.84	0.16	5.92	0.15	-	-	-	-
100	-	-	-	-	2.96	0.15	6.47	0.16	-	-	-	-

TABLE V. STATISTICAL ANALYSIS OF DIFFERENCES BETWEEN INSTRUMENTATION AND LOG ANALYSIS (EFFECT IN MS, P-VALUES)

#TC	S1-delta	S1-p	S2-delta	S2-p	T1-delta	T1-p	T2-delta	T2-p	T3-delta	T3-p	T4-delta	T4-p
1	1132	1.91×10^{-6}	989	1.91×10^{-6}	2473	1.91×10^{-6}	1674	1.91×10^{-6}	1891	1.91×10^{-6}	2691	9.56×10^{-5}
2	2017	1.91×10^{-6}	1665	1.91×10^{-6}	4196	1.91×10^{-6}	2943	1.91×10^{-6}	3563	9.56×10^{-5}	3002	1.91×10^{-6}
3	2159	1.91×10^{-6}	2461	9.56×10^{-5}	4082	1.91×10^{-6}	3535	1.91×10^{-6}	4352	1.91×10^{-6}	4662	1.91×10^{-6}
4	2392	9.56×10^{-5}	2740	1.91×10^{-6}	5220	9.56×10^{-5}	6454	1.91×10^{-6}	5966	1.91×10^{-6}	7174	1.91×10^{-6}
5	2538	1.91×10^{-6}	2841	1.91×10^{-6}	5436	1.91×10^{-6}	7918	1.91×10^{-6}	7751	1.91×10^{-6}	6456	1.91×10^{-6}
6	3537	1.91×10^{-6}	3312	1.91×10^{-6}	6984	1.91×10^{-6}	6968	1.91×10^{-6}	9160	9.56×10^{-5}	8712	1.91×10^{-6}
7	3998	1.91×10^{-6}	6352	1.91×10^{-6}	7977	1.91×10^{-6}	9106	1.91×10^{-6}	10581	1.91×10^{-6}	8423	1.91×10^{-6}
8	5616	1.91×10^{-6}	4295	9.56×10^{-5}	8790	1.91×10^{-6}	9266	1.91×10^{-6}	10355	1.91×10^{-6}	11405	1.91×10^{-6}
9	5152	9.56×10^{-5}	6456	9.56×10^{-5}	8927	1.91×10^{-6}	12782	1.91×10^{-6}	13983	1.91×10^{-6}	12754	1.91×10^{-6}
10	5685	1.91×10^{-6}	5949	1.91×10^{-6}	9996	1.91×10^{-6}	13258	1.91×10^{-6}	13927	1.91×10^{-6}	13961	1.91×10^{-6}
25	13655	1.91×10^{-6}	11847	1.91×10^{-6}	22006	1.91×10^{-6}	32076	1.91×10^{-6}	34087	9.56×10^{-5}	32649	1.91×10^{-6}
50	-	-	28131	1.91×10^{-6}	42555	1.91×10^{-6}	59645	1.91×10^{-6}	-	-	-	-
75	-	-	28981	3.81×10^{-6}	63135	1.91×10^{-6}	87869	1.91×10^{-6}	-	-	-	-
100	-	-	-	-	83471	1.91×10^{-6}	119677	1.91×10^{-6}	-	-	-	-

lower slope, i.e., the overhead increases much less for every additional test activity than instrumentation does. Again, the linear increase depends on the process. The initial penality for log analysis (i.e., test activity count is 0) can be estimated by the linear fitting to approximately 0.5s, which is the expected value: Due to the 1s write delay of the event log the average waiting time for all events to be written to the database is 0.5s.

The different slopes of instrumentation and log analysis lead to different relative overhead as shown in Figure 10. Because the the overhead of instrumentation increases more than the unmeasured test duration, the relative overhead increases when more test activities are executed. In contrast, log analysis increases slower. This results in a decreasing – or for one process nearly constant – relative overhead. However, the relative overhead does not increase or decrease linearly. A logarithmic regression model provides a good fit.

D. Interpretation

1) RQ1: Overhead of Instrumentation: Our measurements for the overhead of instrumentation is in line with already published metrics [13]: The absolute overhead ranges between 1656ms and 122600ms. However, the overhead increases with larger test suites. Thus, the relative overhead increases from 67% to 647% for large test suites. In practice this overhead is considerably large. For nightly builds even a 200% increase of test time would in many environments be deemed impractical. Also research projects, which execute many test suites, e.g., for evaluating different test generation approaches, are impacted heavily.

2) RQ2: Overhead of Log Analysis: Our measurements for the overhead of log analysis demonstrate that the absolute overhead increases and the relative overhead decreases with more test cases. The maximum absolute overhead of 4.5s for

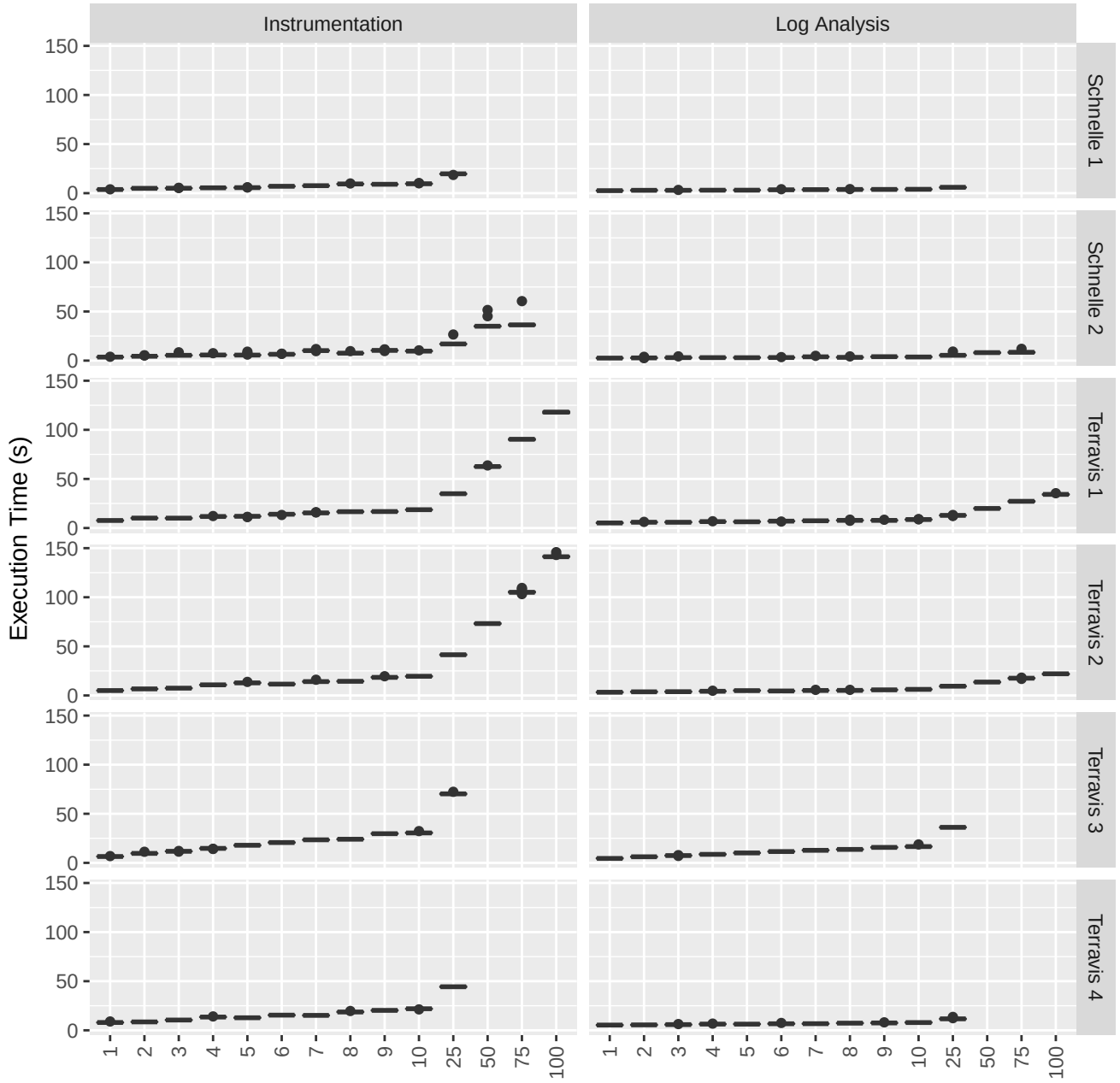

Figure 6. Overall Execution Times (s)

100 test cases the performance penalty is little. This means that measuring approx. 130 test suites of such size would only impose a ten minute overhead (e.g., during nightly builds.) This overhead is much more tolerable in industry projects and research projects, which execute many test suites.

3) RQ3: Relationship between Overhead of Instrumentation and Log Analysis: Our measurements clearly show that log analysis is significantly faster than instrumentation. In the improved version presented in this article, this is also true for trivial test suites, i.e., test suites with only one test case, which were sometimes slower [1] in the unoptimized original version.

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 required in order to evaluate typical overhead ranges. Unit test suites for an executable business process in Terravis contain between 1 and 296 test cases. On average a business process is covered by 27.7 test cases. If we take our measurements for 25 test cases

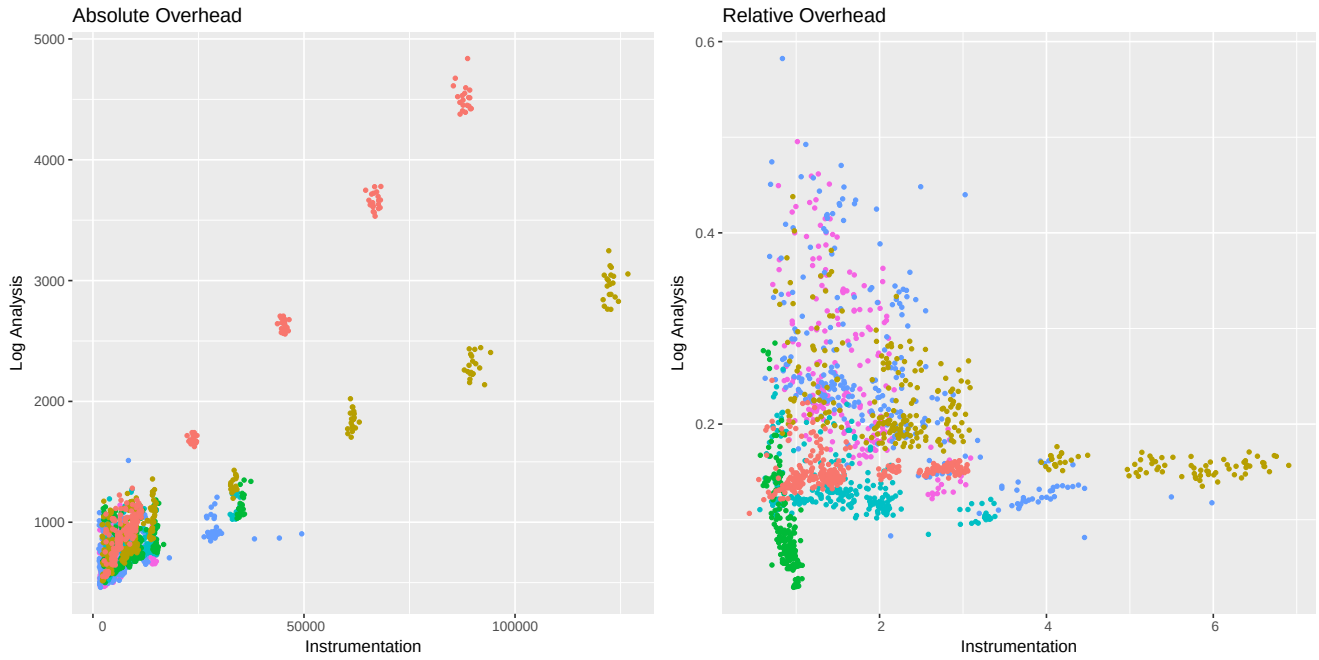

Figure 7. Coverage Measurement Overhead

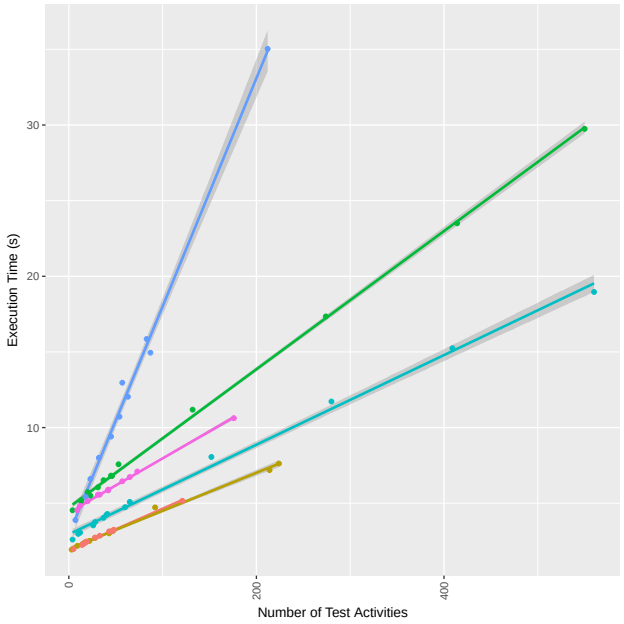

Figure 8. Relationship between Test Suite Execution Time and Number of Test Activities

as a reference the relative overhead is between 24% and 55% for log analysis while it already is between 144% and 268% for instrumentation. This means log analysis has a huge performance benefit when measuring test coverage.

4) RQ4: Influence of Test Suite Size on Instrumentation Overhead: The absolute overhead of an instrumented test run increases linearly with the number of test activities contained in the test suite. However, the linear increase is larger than the linear increase of a normal test run. Therefore, relative overhead of test suite execution time increases non-linearly with an increasing number of test activities. As a result, the instrumentation method does not scale: The larger test suites are getting, the larger the absolute and relative overhead gets. Especially when using test coverage to assess the quality of generated tests, which can easily generate large test suites, the bad scalability will need to be dealt with. For example, test execution needs to be parallalized much sooner than would be otherwise necessary.

5) RQ5: Influence of Test Suite Size on Log Analysis Overhead: Like with instrumentation, the absolute overhead of log analysis increases linearly with the number of test activities. However, the slope is less. The relative increase there gets not-linearly less with more test activities executed as part of a test suite run. Therefore, log analysis can better scale with larger test suites.

E. Threats to Validity

As with every empirical research there are associated threats to validity. While we could increase the number of processes from our initial study [1], our sample mainly consists of processes developed in one project. Thus, 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 all test suite sizes is so low that we are confident that replications will find the same results.

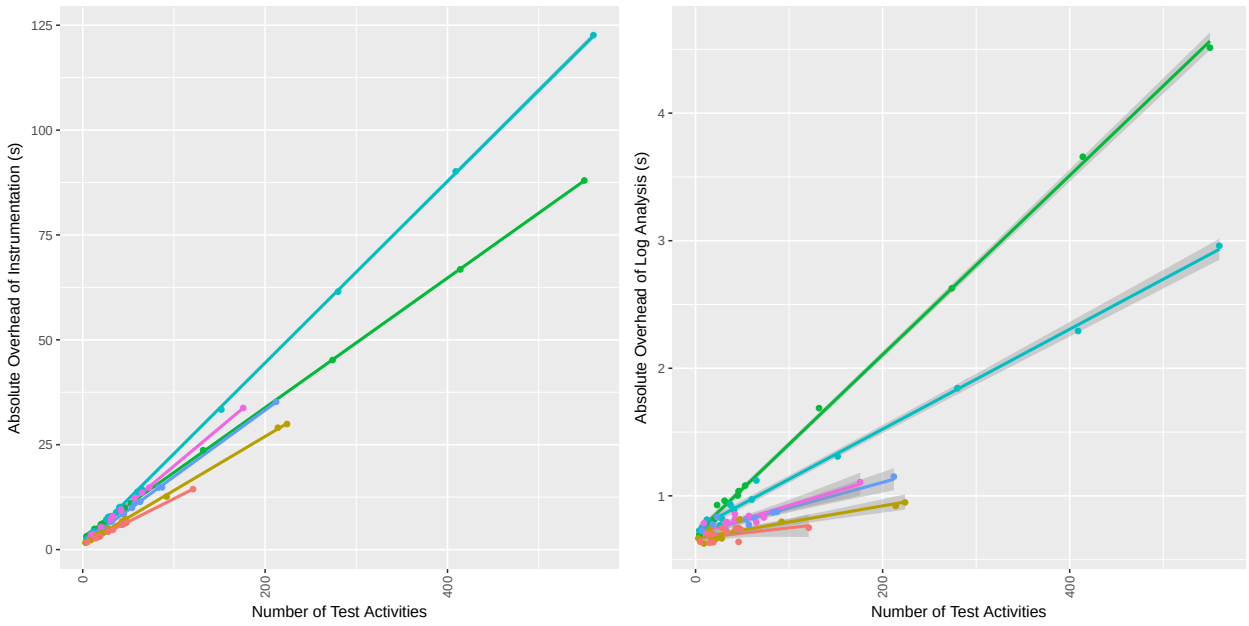

Figure 9. Relationship between Test Suite Execution Time and Absolute Overhead of a) Instrumentation and b) Log Analysis

However, our sample is also constrained to one process engine. This poses the threat of non-portability to other process engines. Therefore, we analyzed Apache ODE, which is an open-source BPEL engine, and found that it emits all necessary events as well [37].

As long as the process engine stores all relevant events that are required for calculating the test coverage metrics, the log analysis can be implemented for such a process engine. 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 despite the claim that this approach is portable: While BPEL is standardized, its extensions and especially the deployment artefacts are not as we encountered when we tried to measure industry projects.

The presented numbers are clearly 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 much smaller: Manual tests take longer for executing the same number of processes, because user interactions require time, which makes the process duration longer. Thus, we do not think that the relative overhead can be generalized to other test types. Even automated integration and system tests are slower because real services usually respond much slower than mocked services that reply a predefined message. For example, automated acceptance tests written for Behavior-Driven Development [38] in the same project take up to 2 minutes to complete per test case [39].

Another threat is the presence of configuration options that heavily impact performance: In the case of the used BPMS - Informatica ActiveVOS - the configurable write delay of the event log can impose longer waiting times. Therefore, environments with a higher configured delay will experience worse log analysis performance because the event log is not immediately available after unit tests are completed. When configured extremely enough, this can lead to a worse performance of log analysis compared to instrumentation.

VII. Conclusion & Future Work

Within this article we evaluated 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. We demonstrated that the new approach utilizing log analysis is significantly faster than the instrumentation approach. With the enhanced waiting strategy for the event log, this is even true for small test suites, which was not the case in the original version.

Furthermore, the log analysis approach can be used in more scenarios than the instrumentation approach: Because all activities for mining 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 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

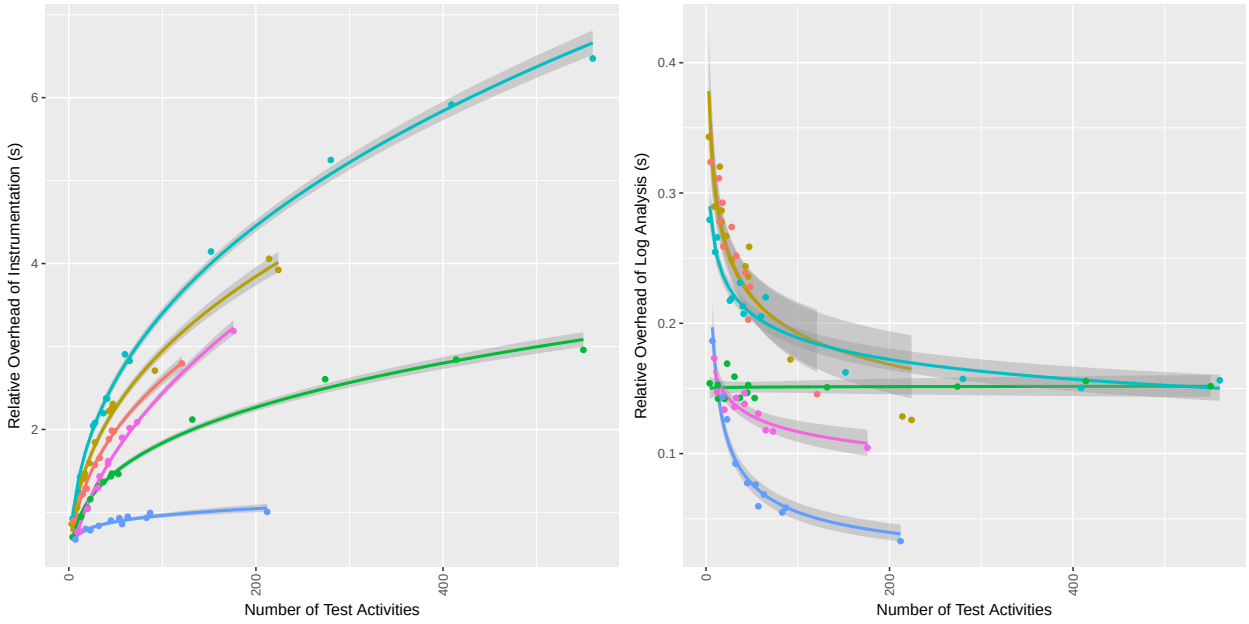

Figure 10. Relationship between Test Suite Execution Time and Relative Overhead of a) Instrumentation and b) Log Analysis

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.

Our research implementation is available as open source and is free to use for both researchers and practitioners. Being able to quickly and easily calculate test coverage for many test types allows further research into executable process test methods, e.g., experiments on the influence of different testing approaches on test coverage.

Acknowledgment

This research was not funded by any institution or research grant. The author is neither affiliated with Informatica nor involved in the development of ActiveVOS in any way. The author is part of the Terravis development team.

References

[1] D. Lübke, "Calculating Test Coverage for BPEL Processes With Process Log Analysis," in BUSTECH 2018, The Eighth International Conference on Business Intelligence and Technology, 2018, pp. 1–7.

[2] D. Lübke, "Unit Testing BPEL Compositions," in Test and Analysis of Service-Oriented Systems, L. Baresi and E. D. Nitto, Eds. Springer, 2007, ch. Unit Testing BPEL Compositions, pp. 149–171.

[3] E. Cambronero, J. C. Okika, and A. P. Ravn, "Consistency Checking of Web Service Contracts," Int'l Journal Advances in Systems and Measurements, vol. 1, no. 1, 2008, pp. 29–39.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] T. Schnelle, "Generierung von BPELUnit-Testsuites aus Klassifikationsbäumen," Master's thesis, Leibniz Universität Hannover, Fachgebiet Software Engineering, 2016.

[16] 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.

[17] G. M. Kapfhammer, "Automatically and Transparently Distributing the Execution of Regression Test Suites," in Proceedings of the 18th International Conference on Testing Computer Software, June 2001.

[18] Informatica. BPEL Unit Testing. [Online]. Available: http://infocenter.activevos.com/infocenter/ActiveVOS/v92/index.jsp?topic=/com.activee.bpep.doc/html/UG21.html (2016)

[19] 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)

[20] A. T. Endo, A. da Silva Simao, S. d. R. S. de Souza, and P. S. L. de Souza, "Web services composition testing: a strategy based on structural testing of parallel programs," in Testing: Academic & Industrial Conference-Practice and Research Techniques (taic part 2008). IEEE, 2008, pp. 3–12.

[21] B. Miranda, "A Proposal for Revisiting Coverage Testing Metrics," in Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 899–902. [Online]. Available: http://doi.acm.org/10.1145/2642937.2653471

[22] C. Bartolini, A. Bertolino, S. Elbaum, and E. Marchetti, "Whitening SOA Testing," in Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 161–170. [Online]. Available: http://doi.acm.org/10.1145/1595696.1595721

[23] M. M. Eler, A. Bertolino, and P. C. Masiero, "More testable service compositions by test metadata," in Proceedings of 2011 IEEE 6th International Symposium on Service Oriented System (SOSE), Dec 2011, pp. 204–213.

[24] 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.

[25] K. Kaschner and N. Lohmann, "Automatic test case generation for interacting services," in International Conference on Service-Oriented Computing. Springer, 2008, pp. 66–78.

[26] S. Ji, B. Li, and P. Zhang, "Test Case Selection for Data Flow Based Regression Testing of BPEL Composite Services," in Services Computing (SCC), 2016 IEEE International Conference on. IEEE, 2016, pp. 547–554.

[27] M. D. Weiser, J. D. Gannon, and P. R. McMullin, "Comparison of Structural Test Coverage Metrics," IEEE Software, vol. 2, no. 2, Mar 1985, pp. 80–85, copyright - Copyright IEEE Computer Society Mar/Apr 1985; Last updated - 2014-05-17; CODEN - IESOEG. [Online]. Available: https://search.proquest.com/docview/215840674?accountid=14486

[28] Q. Yang, J. J. Li, and D. M. Weiss, "A survey of coverage-based testing tools," The Computer Journal, vol. 52, no. 5, 2009, pp. 589–597.

[29] A. Mockus, N. Nagappan, and T. T. Dinh-Trong, "Test coverage and post-verification defects: A multiple case study," in 2009 3rd International Symposium on Empirical Software Engineering and Measurement, Oct 2009, pp. 291–301.

[30] C. Pavlopoulou and M. Young, "Residual Test Coverage Monitoring," in Proceedings of the 21st International Conference on Software Engineering, ser. ICSE '99. New York, NY, USA: ACM, 1999, pp. 277–284. [Online]. Available: http://doi.acm.org/10.1145/302405.302637

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

[32] S. Schönig, M. Seitz, C. Piesche, M. Zeising, and S. Jablonski, "Process observation as support for evolutionary process engineering," International Journal on Advances in Systems and Measurements Volume 5, Number 3 & 4, 2012, 2012.

[33] M. Jäntti, A. Cater-Steel, and A. Shrestha, "Towards an improved it service desk system and processes: a case study," International Journal on Advances in Systems and Measurements, vol. 5, no. 3 & 4, 2012, pp. 203–215.

[34] E. Bruballa Vilas, Á. Wong, D. I. Rexachs del Rosario, E. Luque, and F. Epelde Gonzalo, "Evaluation of response capacity to patient attention demand in an Emergency Department," International journal on advances in systems and measurements, vol. 10, no. 1&2, 2017, pp. 11–22.

[35] 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. 2-3, 2007, pp. 162–198.

[36] D. Lübke, A. Ivanchikj, and C. Pautasso, "A Template for Sharing Empirical Business Process Metrics," in Business Process Management Forum - BPM Forum 2017, 2017.

[37] Apache Software Foundation, "ODE Execution Events," 2018. [Online]. Available: http://ode.apache.org/ode-execution-events.html, lastaccessed2018-08-07

[38] D. North, "Introducing BDD," 2006, http://dannorth.net/introducing-bdd. [Online]. Available: http://dannorth.net/introducing-bdd

[39] D. Lübke and T. van Lessen, "Modeling Test Cases in BPMN for Behavior-Driven Development," IEEE Software, vol. Sep/Oct 2016, Sep/Oct 2016, pp. 17–23.