Calculating BPEL Test Coverage through Instrumentation

Daniel Lübke, Leif Singer
Leibniz Universität Hannover
Software Engineering Group
Welfengarten 1, D-30167 Hannover, Germany
{daniel.luebke, leif.singer}@inf.uni-hannover.de

Alex Salnikow
Leibniz Universität Hannover
Software Engineering Group
Welfengarten 1, D-30167 Hannover, Germany
alex.salnikow@se.uni-hannover.de

Abstract

Assessing the quality of tests for BPEL processes is a diffcult task in projects following SOA principles.

Since insufficient testing can lead to unforeseen defects that can be extremely costly in complex and mission critical environments, this problem needs to be addressed.

By using formally defined test metrics that can be evaluated automatically by using an extension to the BPELUnit testing framework, testers are able to assess whether their white box tests cover all important areas of a BPEL process.

This leads to better tests and thus to better BPEL processes because testers can improve their test cases by knowing which important areas of the BPEL process have not been tested yet.

1 Introduction

With the rise of Service-Oriented Architecture (SOA) and the Web Services stack of standards, the IT departments of many companies are moving to newly built and flexible architectures. (Web) Services are being developed which can be used in so-called compositions in order to realize fully-automated business processes. Such processes operate at the heart of an enterprise. Therefore, the technical implementation becomes mission-critical.

The WS-BPEL (short BPEL) standard [4] is a language providing the means to implement service compositions. Because BPEL is standardized by OASIS and backed by many software vendors, it is often chosen by enterprises. Before putting BPEL processes and the corresponding Web Services into use, these systems must be tested intensively in order to mitigate the risk imposed by defects due to their important role in the enterprise.

However, Web Services and BPEL both are relatively new technologies, and many methods that are available for general software development have either not been ported to SOA yet or SOA-related experience is still missing.

Within this paper, we describe test coverage metrics for BPEL processes and an instrumentation strategy as a way of measuring them. Test coverage is a class of metrics determining how much of the software has actually been executed by a set of test cases. It is often used as an indicator for the quality of the tests: Areas which are not being executed by any test at all are more likely to contain errors than those which have been tested.

In the next section of this paper, we will outline related work from the field of test coverage measurement and metrics before we will introduce test coverage metrics for BPEL. Afterwards, we introduce how BPEL processes can be instrumented in order to measure the test coverage. The case study, described in the subsequent section, delivers some experiences we use to highlight implications and achievements of our approach. Finally, we summarize the implications and achievements of our approach and provide conclusions as well as a possible outlook.

2 Related Work

Metrics for measuring software and test quality are common in the field of Software Engineering. Since BPEL is "programming in the large", it makes sense to adopt these metrics in the new field of SOA as well.

An example of a successfully adopted metric is complexity: McCabe defined Complexity [19] for software programs which has been modified by Cardoso [8] for usage in BPEL.

For measuring test quality, many test coverage metrics based on different coverage subjects have been proposed and are used in many development projects:

Code Coverage Metrics (like Statement Coverage [6, p. 44], Branch Coverage [6, p. 45], Conditional Coverage [6, pp. 44]) measure how much of the actual program is executed. When achieving 100%, each statement, branch or path is executed at least once. This

metric can only be used when doing glass box tests, e.g. unit tests by the developer.

Requirements Coverage Metrics measure which percentage of the requirements have been tested and deemed functional. Such metrics are often used in acceptance and system tests, e.g. see [15].

Data Coverage Metrics measure how and when variables have been used in a certain manner, e.g. see [7].

Within this paper, we will deal with code coverage metrics that can be used to judge the quality of white box tests.

To calculate test coverage, the parts of the code that are actually being executed need to be traced. Tools like Cobertura [2] instrument the code in a way that allows the framework to track the execution path taken by a suite of tests.

In the field of BPEL unit testing, approaches similar to the one presented in this paper have already been published. The following paragraphs point out notable differences.

[5] employs a transition coverage, similar to the link coverage this paper introduces, for generating test cases for a BPEL process, but does not define it explicitly. A transition coverage of 100% seems to be achievable if each transition has been executed at least once. This is not a useful definition for assessing the quality of unit tests, though: as we will mention in section 3.3, this approach would result in skewed results, e.g. due to BPEL-specific mechanisms like dead path elimination.

In [14], the authors discuss their approach to testing the XPath parts of BPEL processes. Their solution generates test cases for BPEL processes and combines them into test suites using a strategy that maximizes coverage. This coverage seems to be based on well-defined criterions, but again is only concerned with XPath. The paper briefly touches the issue of instrumentation for coverage measurements, but does not address it in detail.

Yan, Li, Yuan, et al. use concurrent path analysis and and a constraint solver in [20] to generate suitable sets of test cases for BPEL processes. To reduce the number of test cases, their paper defines two simple coverage metrics. They do not suggest to use the coverage metrics for improving manual test case creation. This is still required, as their approach omits some BPEL constructs, such as compensation handlers and nested scopes, which are covered in this paper.

3 Test Coverage Metrics

When writing unit tests, determining the relevance of one's tests is a crucial part of the testing process. Running many unit tests successfully has no meaning by itself – if they touch only a small fraction of the code, it is dangerous to assume the rest of the code to be without defects. Therefore, developers and software testers employ a few important metrics to measure test significance.

Another important aspect to take into account when writing tests is efficiency. For most services, the number of possible test inputs is infinite. Therefore, test writers should choose their tests and test inputs in a manner that tries to maximize the utility of tests and at the same time minimizes the number of test runs. There are several known strategies to write such tests and input values, e.g., equivalence class partitioning. But again, testers cannot know the actual efficiency of their tests without knowing what parts of the code under test were actually executed in a test run. This is another case in which test metrics are an essential tool for test writers.

One class of these metrics is code coverage. Each of the code coverage metrics measures the amount of code a test actually executes. Statement Coverage, for example, is the ratio of the number statements executed by a test by total number of statements. Branch Coverage, however, is defined as the number of branches executed during a test run divided by the total number of all possible branches in the control flow graph given by the code under test.

Both metrics are highly useful when writing unit tests, as they point out to the test writer which parts of the code under test are not yet covered by their tests. This allows for a systematic approach to improving existing unit tests, thus providing an opportunity to increase overall code quality.

While code coverage metrics have successfully been used in mainstream software development and appropriate tools for measurement are widely deployed, there currently is no agreed-upon method to assess the quality of unit tests for executable business processes written in BPEL. Employing a formal description of BPEL processes – described in further detail in [3] and based on [1] – this paper introduces definitions for five code coverage metrics for BPEL processes.

More specifically, we first define Activity Coverage for BPEL, which is based on Statement Coverage known from existing approaches. Second, we present Branch Coverage for BPEL, which is also an equivalent for the already widely used coverage metric with the same name.

To account for the differences between classic programming paradigms and those present in BPEL, we continue by introducing three coverage metrics specifically targeted at BPEL. We define Link Coverage to address the traversal of link elements in BPEL's flow elements, and two Handler Coverage metrics to measure the execution of fault and compensation handler elements, two important error handling mechanisms found in BPEL.

These three additional coverage metrics are required for BPEL, since they address the workflow patterns not covered by traditional coverage metrics. Especially the flow

construct with its links, transition conditions and join conditions that provide a parallel execution path that's being decided upon in a local manner, stand out in this regard.

3.1 Activity Coverage

Statement Coverage, as known from classic software development, is the ratio of the number of statements executed in tests and the number of statements overall. Since the basic activities of a BPEL process are quite similar to the statements found in regular program code, we employ the idea of the Statement Coverage to create a BPEL-specific Activity Coverage. For a formal definition, we first introduce some fundamental sets. For more details, please refer to [3].

- T is the set of all activity types that are defined in BPEL, e.g. "flow", "pick", or "invoke".
- $T_B \subset T$ is the set of all *basic* activity types that are defined in BPEL, e.g. "invoke" or "receive".
- W is a given BPEL process.
- $A \in W$ is the set of all activities present in a given BPEL process W.
- $A^{basic} \subseteq A$ is the set of all *basic* activities present in a given BPEL process.
- $A_t^{executed} \subseteq A^{basic}, t \in T_B$ is the set of all *basic* activities of type t that have been executed during testing.

For a single activity type t and a BPEL process W, we can thus define Activity Coverage to be

$S_t(W) := \frac{|A_t^{executed}|}{|A_t|},\ t \in T_B$

Based on $S_t(W)$, it now makes sense to define the overall Activity Coverage for a BPEL process W as

Definition 3.1 *Activity Coverage*

$$S(W) := \frac{\left|\bigcup_{t \in T_B} A_t^{executed}\right|}{|A^{basic}|}$$

While it is advisable to maximize the Activity Coverage when testing a BPEL process, it is still a weak indicator. More complex activities, like loops and conditionals, are not yet taken into account. We therefore continue by providing a stronger indicator for test coverage, namely a BPEL-specific Branch Coverage.

3.2 Branch Coverage

In the classic Branch Coverage metric, the number of executed code branches are being compared to the total number of possible code branches. For our BPEL-specific variant, we will do the same. To be able to do so, we need to define how exactly these numbers can be calculated for a given BPEL process. We do so by providing two functions β and $\beta^{executed}$ for each activity type defined in BPEL to calculate the number of all branches and the number of branches executed in tests. For the details, we again refer to [3].

Since the process element is the root activity in all BPEL processes, we can now use the β functions for the process element to give the definition for Branch Coverage in BPEL as follows:

Definition 3.2 *Branch Coverage*

$$B(W) = \frac{\beta_{scope}^{executed}(process)}{\beta_{scope}(process)}$$

3.3 Link Coverage

Within BPEL's flow element, developers may create sophisticated control flows using links that connect activites. Each link may have a transition condition determining at runtime whether that link may be followed. Also, each activity that is the target of one or more links may have a join condition further determining control flow at runtime. Considering further the possibly parallel execution of activites linked using these language features, it should be obvious that nontrivial behaviour may easily result. This is especially important to test thoroughly. Therefore, this section adds a BPEL-specific Link Coverage to further enhance the coverage metrics already presented.

Join conditions placed inside an activity can only use boolean expressions and the status values of the activity's links. Therefore, it is sufficient for the metric to only consider transition conditions. We construct the metric in such a way that, for a suite of tests to achieve a 100% Link Coverage, requires each transition condition to have been true and false at least once in a test suite.

To ensure the metric will provide practical utility to testers, we further introduce these additional restrictions:

- Because transition conditions with a constant value cannot be influenced by the input data of a test, the metric only considers conditions including at least one variable.
- Only transition conditions that are actually being evaluated during a test are being counted in.
- Transition conditions that have been set to *false* by BPEL's dead path elimination will also not be counted.

Without these restrictions, a Link Coverage of 50% could be easily achieved by running some rather simple tests without actually requiring the testing of the transition conditions.

For the mathematical definition, please see [3]. To preserve clarity, we will only present a natural language description of the metric in this paper.

Definition 3.3 *Link Coverage*

The Link Coverage for a BPEL process tested by a test suite is the sum of

- *The number of links with a variable transition condition that have been evaluated to* true *at least once in a test, divided by twice the number of links with a variable transition condition present in the process under test; and*
- *the number of links with a variable transition condition that have been evaluated to* false *at least once in a test, divided by twice the number of links with a variable transition condition present in the process under test.*

The division by two is needed because each link is required to have at least once been evaluated to *true* and *false* as well – in different tests, of course.

With link coverage, it is possible to better assess the effects of tests in flow environments. However, no coverage metric presented in this paper will consider the order in which activities may be executed in flow environments.

3.4 Handler Coverage

To handle errors and exceptions in a running business process, BPEL provides the concepts of fault handlers and compensation handlers. Fault handlers provide the means to react to actual faults, while compensation handlers are meant to undo the successful actions of a scope that failed executing completely. Since BPEL processes, just like the business process they present, may run for a long time, and crucial business value is often derived from them, high-quality testing of fault behaviour is essential. We again present these metrics in natural language, leaving the mathematical details to [3].

Definition 3.4 *Fault Handler Coverage*

The Fault Handler Coverage for a given BPEL process tested by a given test suite is the number of catch and catchAll blocks executed in one or more tests divided by the overall number of catch and catchAll blocks present in the BPEL process.

Definition 3.5 *Compensation Handler Coverage*

The Compensation Handler Coverage for a given BPEL process tested by a given test suite is the number of compensation handlers executed in one or more tests divided by the overall number of compensation handlers present in the BPEL process.

With these five coverage metrics defined, a facility to measure and calculate them must be provided to be able to integrate them into a tester's workflow. Since instrumentation is a proven strategy to measure coverage metrics, we will employ it in unit testing for BPEL as well. The following section provides more detail on this.

4 Instrumentation of BPEL Processes

To calculate the metrics defined above, we added instrumentation capabilites to BPELUnit [13], a test framework for BPEL processes. While there are other approaches to collecting similar data – e.g., tracing by using a BPEL debugger – we found instrumentation to be the most vendor-neutral method: The instrumented BPEL process may still run in any BPEL engine it was able to run in before. Since the debugging APIs for tracing vary from vendor to vendor, the solution would have to be customized for each BPEL engine providing such an API.

Instrumentation is a wide-spread technique that adds data collection statements into the code under test. In our case, this was the addition of invoke activites to the BPEL process under test which would call a coverage logging Web Service when executed by the BPEL engine. The logging service, which is run by the BPELUnit framework, would then be notified of the execution of a specific activity, identified by a unique marker.

Our instrumentation can be divided into three phases:

1. Analysis of the BPEL process to determine the activities relevant for coverage measurement.
2. Insertion of marker comments into the BPEL process; specific for each metric.
3. Replacement of the markers with invocations of the coverage logging Web Service.

The following sections describe these three phases, starting with the analysis of the process.

4.1 Analysis of the BPEL Process

In this phase, all elements of the BPEL process relevant for the calculation of the coverage metrics are being determined. These are:

- Activity Coverage: all basic activities;
- Branch Coverage: all structured activities;
- Link Coverage: all links with a non-constant transition condition;
- Fault Handler Coverage: all catch and catchAll elements;
- Compensation Handler Coverage: all compensation handlers.

The results of this preliminary analysis are saved for use during the other two phases. This ensures the integrity of the BPEL process when modifying it in the subsequent phases, as it prevents these modifications to influence both the measurement as well as the semantics of the process.

4.2 Insertion of Markers

This phase adds marker comments to each activity determind in the first phase. Each marker can be identified uniquely and belongs to a certain metric. Also, each marker identifies an element of control flow relevant for the metric covered by the marker. It is notable that the order in which markers are being inserted is irrelevant, as the original elements have been preservered in the analysis phase.

Each marker now represents the stimulation of a certain element of the control flow. We will now describe for each metric how the markers will be inserted into the BPEL process.

4.2.1 Activity Coverage

With the exception of the receive activity, a marker is inserted before each basic activity. The marker for the receive activities needs to be placed *after* the activity, as it only gets executed when given control by a message sent to the BPEL process. The markers for all other basic activities can be inserted in a way that ensures the execution of the activities right after the marker has been reached.

For basic activities in a sequence activity, the marker can be inserted directly after / before the activity. The sequence activity's semantics take care of the correct execution order by itself.

Basic activities contained in other structured activities, such as loops, can be wrapped in a sequence activity without altering the control flow's semantics. Once wrapped, the insertion of the marker can make use of the sequence activity's semantics again and take place right after / before the basic activity that is to be instrumented.

To insert markers for basic activities in a flow element, i.e., those with incoming and / or outgoing links, the markers may not merely be connected by additional links. Because each link may have an associated transition condition, and activities may contain join conditions, the execution of the activity would not be guaranteed. E.g., a marker might be reached, but the basic activity it monitors does not get executed because of a transition or join condition evaluating to *false*. Also, a link may have only one target and one source, which eliminates the possibility of merely reusing the already existing links.

Thus, our solution is to wrap each of these linked basic activities in a sequence activity. The marker can then be inserted after / before the basic activity, included in the sequence. Again, this preserves the semantics of the BPEL process, as the sequence activity takes care of the execution order itself.

4.2.2 Branch Coverage

In the previous phase, the BPEL process has already been analyzed. Thus, all branches have been identified already. Basically, the markers for Branch Coverage are being inserted following the strategy presented for Activity Coverage, i.e., a sequence activity is inserted along with the marker if needed to preserve the semantics of the BPEL process. Some peculiarities must be taken care of, though:

- **If Activity**: If there is no explicit else branch given in the BPEL process, it must be inserted along with an accompanying marker.
- **Flow Activity**: A flow activity may start with a receive activity that instantiates the BPEL process. In this case, the mere wrapping of the receive activity in a sequence activity with a marker inserted before the receive activity would create an invalid BPEL process, as the first activity must always be a receive activity with the createInstance attribute set to *yes*. Again, we must make sure that the marker succeeds the receive activity by wrapping it in a sequence, but inserting the marker *after* the receive activity.
- **RepeatUntil Activity**: The repeatUntil activity is a loop activity that evaluates its condition *after* each iteration. Thus, there are two branches that need to be logged: the first execution of the loop, as well as any subsequent executions. Therefore, we need to wrap the loop's contents in a sequence activity and then insert two distinct logging markers into that sequence. The first marker monitors the first execution and is the first activity in the inserted sequence. The second marker needs to be wrapped in an if activity determining whether the loop will be executed again. Also, an additional counter variable must be introduced for the if activity's condition to be correct.

4.2.3 Link Coverage

To collect the data required for calculating the Link Coverage metric, markers must be inserted that monitor the transition conditions found in flow activities of the BPEL process. By definition of the metric, both the actual conditions as well as their negated counterparts need to be logged. Special care must be taken of boundary crossing problems when modifying flow activites: In BPEL, several paths are not allowed to be taken by links, e.g. entering a loop activity from outside of it.

To avoid this problem, we insert another flow activity into the process, wrapping the source activity of the link that is to be monitored. We then add a copy of the link, pointing from the source activity to a marker for the coverage logging Web Service, as well as a negated copy of

the link, also pointing from the source activity to a marker. This way, both *true* and *false* evaluations of the link can be logged.

4.2.4 Fault and Compensation Handler Coverage

Finally, we need to insert markers for the Fault and Compensation Handler Coverage metrics. We insert them into the handlers by using a wrapping sequence activity again, which will then contain the logging marker, followed by the handler's activities.

Since BPEL allows the use of inline handlers for invoke activities, these must first be converted to a regular scope. That scope then contains the inline handlers, followed by the original invoke activity. After this conversion, the handlers can be treated just like regular handlers.

This completes the description of the second phase. The inserted markers now have to be replaced with actual invocations of the coverage logging Web Service.

4.3 Replacement of Markers with Invocations

Since the previous phase already took care of preserving the syntax and the semantics of the BPEL process, this phase merely needs to replace the inserted markers with activities that invoke the coverage logging Web Service, transmitting the ID of the marker that was replaced.

After tests on an instrumented BPEL process have run, BPELUnit calculates the metrics presented in this paper and displays them to the tester. This allows for an immediate optimization of the current tests to improve test coverage.

5 Case Study

To demonstrate the functionality and the value of the metrics described in this paper, we utilized the results of a former project complete by a team of ten computer science students. The project's goal was to develop a student thesis management system for the university's staff. Students' theses, like Bachelor's and Master's theses, can be created, managed, and tracked with this software.

The software consists of a main BPEL process (macroflow) accessing several microflows. The macroflow resembles the actual thesis process from a student's application up to grading the thesis. The microflows compose basic Web Services to bigger building blocks on a business level. The architecture is illustrated in figure 1.

The students also had the task of testing their application. Therefore, two BPELUnit test suites exist, containing 32 test cases for these BPEL processes. One test suite contains the test cases for the macroflow and the other contains the test cases for all the microflows.

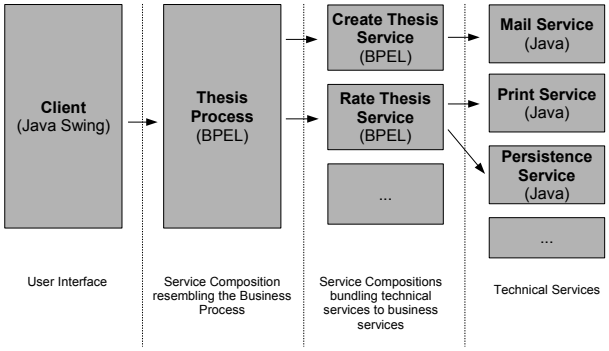

Figure 1. Architecture of the Case Study

Coverage	Total	Tested	Coverage
Activity Overall	**750**	**468**	**62%**
assign	387	277	71%
compensate	0	0	-
empty	109	91	83%
invoke	56	56	100%
receive	21	21	100%
reply	55	23	41%
terminate	14	0	0%
throw	98	0	0%
wait	10	0	0%
Branch Overall	**1020**	**704**	**69%**
Link Overall	**8**	**6**	**75%**
true	4	3	75%
false	4	3	75%
Handler Overall	**64**	**0**	**0%**
Fault	64	0	0%
Compensation	0	0	-

Table 1. Test Coverage of the Macroflow

The test coverage, which was measured after the project had been completed, is given in table 1 for the macroflow and in table 2 for the microflow.

The test coverage clearly indicates that there are areas which have not been tested. Especially fault handlers and compensation handlers, i.e., the error handling, have not been tested at all. Because the testing was not being supported by test coverage metrics, this testing behaviour might be typical for non-student projects as well. However, neglecting error conditions in distributed systems like SOA applications can be fatal. Test coverage reports can show testers the shortcomings of their tests in these areas, and quality managers can use them to guide testers to test error conditions, like limited or missing availability of services in order to test and improve the error handling routines.

By using these BPEL processes as real samples for

BPEL test suites, the performance impact of the instrumentation and the associated Web service calls during test runs can be measured and assessed. The time needed for executing the test suites is shown in table 3. The instrumentation nearly triples the time needed for executing the first test suite and more than doubles the execution time for the second test suite. The absolute times needed for executing the test suites depend on the computer configuration on which the tests are run. The measurements presented in this paper were run on the following configuration:

- AMD Athlon XP CPU, 2.17 GHz
- 2 GBytes Main Memory
- Windows XP Operating System
- ActiveBPEL 3.0.0 BPEL Server

Testing service compositions is always time-consuming due to the associated network overhead and XML processing. The instrumentation adds to this overhead, and results in even longer test-runs. Thus, the instrumentation should only be used when the reports are needed and should not be enabled by default. This is especially true if the tests have to be run frequently like in test-driven projects.

6 Implications and Achievements

While the theoretical definition of BPEL test coverage is useful for defining metrics independent of any implementation and allows for a comparision of test coverage metrics,

Coverage	Total	Tested	Coverage
Activity Overall	**750**	**378**	**50%**
assign	387	225	58%
compensate	0	0	-
empty	109	85	77%
invoke	56	44	78%
receive	21	12	57%
reply	55	12	21%
terminate	14	0	0%
throw	98	0	0%
wait	10	0	0%
Branch Overall	**1020**	**570**	**55%**
Link Overall	**8**	**2**	**25%**
true	4	1	25%
false	4	1	25%
Handler Overall	**64**	**0**	**0%**
Fault	64	0	0%
Compensation	0	0	-

Table 2. Test Coverage of the Test Suite of the Microflows

Test Suite	Time/Plain	Time/Instr.	Increase
Test Suite 1	24.3s	68.7s	282%
Test Suite 2	14.6s	32.8s	224%

Table 3. Test Run Duration for the Test Suites with and without Instrumentation

these metrics have to be measurable in practice to be able to help software projects.

By using BPELUnit with the coverage extension, it is possible to measure the test coverage of test cases. Instrumentation may be used for collecting data in a platform-independent manner, i.e., it can be used in conjunction with every available BPEL engine. However, instrumentation adds overhead that increases the time needed for test execution by a factor of approximately 2 to 3, as has been shown in our case study. Therefore, test coverage data should only be calculated if the current goal really is either the improvement of existing tests or the assessment of the tests' quality. The speed penality is measurably higher than in traditional programming languages like Java, due to the use of XML messaging.

However, the coverage metrics are extremely valuable. If these results had been available while the project from the case study was still running, we would have been able to better control and influence the test process. In industrial SOA projects, QA teams and project management can now use the metrics for guiding and judging the test process. Because this process can be fully automated, it is possible to create coverage reports for daily builds to regularly check the test progress.

In our case study, testers neglected to deal with error conditions, which is comparable to our experience with students in software projects that do not follow SOA principles. While we have no proof, we expect industrial testers to behave similarily. By having a means for calculating and gathering test coverage data, such situations can be detected more easily and explained better to the development team.

7 Conclusions & Outlook

Within this paper, we presented defintions for BPEL test coverage metrics. In order to measure coverage, several possibilities exist. We described instrumentation as one way to collect all necessary data about the execution of a BPEL process in a platform-independent manner. Instrumentation has been added on top of the BPELUnit framework.

We gathered test coverage data for a student project in which an application had been developed that was mainly dependent on BPEL processes. We calculated and analyzed the test coverage data. With this information, it was easy

to identify the shortcomings of the test cases. In this case, the fault handling had not been tested at all. Although this case study is by no means representative, similar observations in software projects with other architectures suggest that testers have to look for fault handling in SOA projects as well. Test coverage metrics are a good foundation to start from. Test coverage allows to quickly identify fault and compensation handlers that have not yet been tested.

However, calculating test coverage metrics adds overhead. Especially the platform-neutral approach employing instrumentation is time-consuming. Therefore, test coverage metrics should only be calculated when explicitly needed.

Apart from the classical coverage metrics this paper's metrics are based on, several others have been defined in the past, some of them in regular use – e.g., condition coverage and path coverage. Defining them for BPEL and extending our existing instrumentation with new coverage metrics should be easy when taking our results as a basis. For now, we will concentrate on better integration of our results into the BPEL development process, though, since we consider our current set of metrics to be of sufficiently high value to developers and testers.

In a broader context, test coverage metrics are not only useful in industrial projects. Analyzing tests, especially in test-driven projects, like Extreme Programming (XP) [10], is valuable in academia as well and has often been conducted with more traditional languages (e.g. [16, 17, 18, 9]). By having a tool available for this purpose, research may, for example, analyze SOA XP projects. Thus, comparison to "classical" XP projects is possible and the effects of the combination of SOA and XP can be better explored.

References

[1] Chun Ouyang, H. Verbeek, Wil M.P. van der Aalst, Stephan W. Breutel, Marlon Dumas, and Arthur H.M. ter Hofstede. Formal Semantics and Analysis of Control Flow in WS-BPEL. Technical Report BPM-05-13, BPM Center, 2005.

[2] Cobertura Project. Cobertura Homepage. WWW: http://cobertura.sourceforge.net/, last access 2007-09-01, August 2006.

[3] Daniel Lübke and Alex Salnikow. Definition and Formalization of BPEL Process Test Coverage. Technical report, Leibniz Universität Hannover, FG Software Engineering, http://www.se.uni-hannover.de/techreports/2009-01-DefinitionAndFormalizationOfBpelProcessTestCoverage.pdf, 2008.

[4] Diane Jordan, John Evdemon, Alexandre Alves, Assaf Arkin, Sid Askary, Charlton Barreto, Ben Bloch, Francisco Curbera, Mark Ford, Yaron Goland, Alejandro Guízar, Neelakantan Kartha, Canyang Kevin Liu, Rania Khalaf, Dieter König, Mike Marin, Vinkesh Mehta, Satish Thatte, Danny van der Rijn, Prasad Yendluri, and Alex Yiu. *Web Services Business Process Execution Language Version 2.0*. OASIS, April 2007.

[5] J. Garca-Fanjul, J. Tuya, and C. de la Riva. Generating Test Cases Specications for BPEL Compositions of Web Services Using SPIN. In *International Workshop on Web Services Modeling and Testing (WS-MaTe 2006)*, 2006.

[6] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. Wiley & Sons, 2nd edition, June 2004.

[7] J. R. Horgan and S. London. Data flow coverage and the C language. In *TAV4: Proceedings of the symposium on Testing, analysis, and verification*, pages 87–97, New York, NY, USA, 1991. ACM Press.

[8] Jorge Cardoso. Complexity Analysis of BPEL Web Processes. *Software Process: Improvement and Practice Journal*, 12:35–49, 2006.

[9] Kai Stapel, Daniel Lübke, and Eric Knauss. Best Practices in eXtreme Programming Course Design. In *Proceedings of the 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany*, 2008.

[10] Kent Beck. *Extreme Programming Explained*. Addison-Wesley, 2000.

[11] Z. Li, W. Sun, Z. B. Jiang, and X. Zhang. BPEL4WS Unit Testing: Framework and Implementation. In *Proceedings of the IEEE International Conference on Web Services (ICWS05)*. IEEE, 2005.

[12] Z. J. Li and W. Sun. BPEL-Unit: JUnit for BPEL Processes. In *ICSOC 2006, LNCS 4294*, pages 415–426. Springer-Verlag, 2006.

[13] 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*, pages 33–42, New York, NY, USA, 2006. ACM.

[14] L. Mei, W. K. Chan, and T. H. Tse. Data Flow Testing of Service-Oriented Workflow Applications. In *Proceedings of the 30th International Conference on Software Engineering (ICSE 2008)*, pages 371–380, New York, NY, USA, 2008. ACM.

[15] Michael W. Whalen, Ajitha Rajan, Mats P.E. Heimdahl, and Steven P. Miller. Coverage metrics for requirements-based testing. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 25–36, New York, NY, USA, 2006. ACM Press.

[16] M. Mueller and Ol. Hagner. Experiment about test-first programming. *IEE Software*, 149(5):131–136, October 2002.

[17] Reid Kaufmann and David Janzen. Implications of test-driven development: a pilot study. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 298–299, New York, NY, USA, 2003. ACM.

[18] Thomas Flohr and Thorsten Schneider. Lessons learned from an XP Experiment with Students: Test-First needs more teaching. In *Proceedings of the Profes 2006*, 2006.

[19] Thomas J. McCabe. A Complexity Measure. *IEEE Transaction on Software Engineering*, SE-2(4):308–320, 1976.

[20] J. Yan, Z. Li, Y. Yuan, W. Sun, and J. Zhang. BPEL4WS Unit Testing: Test Case Generation Using a Concurrent Path Analysis Approach. In *17th International Symposium on Software Reliability Engineering (ISSRE'06)*. IEEE, 2006.