A Pattern Language for Workflow Engine Conformance and Performance Benchmarking

SIMON HARRER, Distributed Systems Group, University of Bamberg, Germany
JÖRG LENHARD, Department of Mathematics and Computer Science, Karlstad University, Sweden
OLIVER KOPP, Institute for Parallel and Distributed Systems, University of Stuttgart, Germany
VINCENZO FERME and CESARE PAUTASSO, Software Institute, Faculty of Informatics, USI Lugano, Switzerland

Workflow engines are frequently used in the domains of business process management, service orchestration, and cloud computing, where they serve as middleware platforms for integrated business applications. Engines have a significant impact on the quality of service provided by hosted applications. Therefore, it is desirable to compare them and to select the most appropriate engine for a given task. To enable such a comparison, approaches for benchmarking workflow engines have emerged. Although these approaches deal with different quality attributes, i.e., performance or standard conformance, they face many reoccurring design and implementation problems, which have been solved in similar ways. In this paper, we present a pattern language that captures such common solutions to reoccurring problems (e.g., from test identification, benchmarking procedure validation, automatic engine interaction, and workflow execution observation) in the area of workflow engine conformance and performance benchmarking. Our aim is to help future benchmark authors with the pattern language presented in this paper to benefit from our experience with the design and implementation of workflow engine benchmarks and benchmarking tools.

CCS Concepts: •**General and reference → Measurement; Performance;** •**Applied computing → Business process management systems;** •**Software and its engineering → Software design engineering;**

Additional Key Words and Phrases: workflow engines, benchmarking, performance, conformance, patterns

ACM Reference Format:

S. Harrer, J. Lenhard, O. Kopp, V. Ferme, C. Pautasso. 2017. A Pattern Language for Workflow Engine Conformance and Performance Benchmarking. *EuroPLoP* (July 2017), 46 pages.
DOI: 10.1145/3147704.3147705

1. INTRODUCTION

The service-oriented computing paradigm envisions the usage of services to support the development of rapid, low-cost, interoperable, evolvable, and distributed applications [Papazoglou et al. 2008]. An established part of the field of service-oriented computing is the construction of composite services on the basis of message exchanges between lower-level services [Alonso et al. 2004]. This composition is often achieved by capturing the data- and control-flow between message exchanges of several services in a workflow [Peltz 2003]. The workflow is subsequently deployed on a *workflow engine*, which provides the middleware execution platform, context and cross-cutting functionality, message correlation, and many other features to the hosted workflow. Today, several standards for workflow definition [Mili et al. 2010] and a multitude of engines have emerged[1], including implementations by global middleware vendors, open source solutions, research prototypes, and even cloud-based engines. The range of solutions makes it important for users to compare existing engines with the aim of selecting the best engine for their purpose. The problem is that engines are highly complex products, resulting in an equally complex comparison and selection problem [Harrer 2014]. To address this

[1]Lists of engines are maintained at Wikipedia: https://en.wikipedia.org/wiki/List__of__BPEL__engines and https://en.wikipedia.org/wiki/List__of__BPMN__2.0__engines. The second lists has been built in the context of the BenchFlow project, which is one of our primary sources of patterns here.

problem, workflow engine benchmarking approaches have emerged [Geiger et al. 2016a; Ferme et al. 2015]. Several research groups are developing such approaches and tools that target varying quality properties of workflow engines, such as performance [Ferme et al. 2015; Rosinosky et al. 2016; Daniel et al. 2011; Dujmović 2010] or standard conformance [Geiger et al. 2016a].

When developing approaches and benchmarks, the aforementioned research groups are often facing the same problems, regardless of the actual property that lies in the focus of the benchmark. Such common problems are, for instance, how to identify suitable tests or workloads for engine benchmarks, or how to ensure the correctness of test implementations and benchmarking procedures. Moreover, solutions to such common problems are often similar, leading to the unfortunate situation that multiple groups invest significant effort to solve the same problem and to re-implement duplicate solutions. Since proven solutions to reoccurring problems exist and can be inferred from existing engine benchmarks, it is possible to capture these solutions as *patterns*. The notion of patterns originated from the field of architecture [Alexander 1978], where patterns were used to describe reoccurring structures in buildings. Years later, the idea to describe reoccurring structures in the design of software in the form of patterns [Gamma et al. 1995a] had a huge impact on software development. Since then, patterns have been applied in many areas and contexts, and a multitude of pattern catalogs and languages have been published.

Workflow engine benchmarking is an area, where, to the best of our knowledge, patterns are still lacking. The huge momentum in the development of pattern languages has also led to work that theorizes on pattern structure [Kohls 2010; Kohls 2011] and how to write a pattern [Meszaros and Doble 1998]. Here, we build on these works to specify our patterns. Meszaros and Doble [Meszaros and Doble 1998] propose *name*, *problem*, *context*, *forces*, and *solution* as the mandatory elements of patterns. *Examples* and *relations* are considered as optional elements. In our pattern description, we use *name*, *summary*, *context*, *problems*, *forces*, *solution*, *consequences*, *known uses*, and *relations* as elements. By describing such solutions as patterns, it should be possible to reduce the effort for implementing new workflow engine benchmarks and also to ease the communication among benchmark authors through a shared vocabulary.

During the last two decades, there has been a lot of momentum in the development of workflow languages. Prominent examples of such languages are the *Web Services Business Process Execution Language 2.0* (BPEL) [OASIS 2007] or the *Business Process Model and Notation 2.0* (BPMN) [ISO/IEC 2013]. Moreover, new languages are still being developed. For instance, in the area of application management, the current draft of version 1.1 of the TOSCA standard [OASIS 2017] describes a new workflow language. This will trigger the development of new workflow engines, which have to be benchmarked to be able to compare them. The concepts and patterns behind existing benchmarks should be understood in order to implement a new suitable benchmark for such new engines.

This paper is an extension of a first proposal of workflow engine benchmarking patterns [Harrer et al. 2016]. Here, we are building upon some of the previously published patterns, formulate them in a broader and more reusable fashion, and also include additional patterns related to workflow performance benchmarking. We extend the pattern description and add a discussion of forces and consequences. Moreover, we sketch the relationships between the patterns to connect them into a pattern language. Some of the patterns from our first proposal [Harrer et al. 2016] have been excluded from this paper in favor of new patterns. The group of authors has also been extended to include additional researchers from the field of workflow engine performance benchmarking. Since all of the authors have been working on workflow engine conformance and performance benchmarks and tools for several years, we are confident that the presented patterns can help the authors of future standardized workflow engine benchmarks.

The paper is structured as follows. First, we describe the participants and challenges in workflow engine benchmarking in Sect. 2, and we provide an overview of Betsy and BenchFlow, the two projects used as the main source of experience and knowledge to build the pattern language. Thereafter, we outline the structure of the pattern language in Sect. 3. Sect. 4 describes the patterns, which act as a set of alternative and

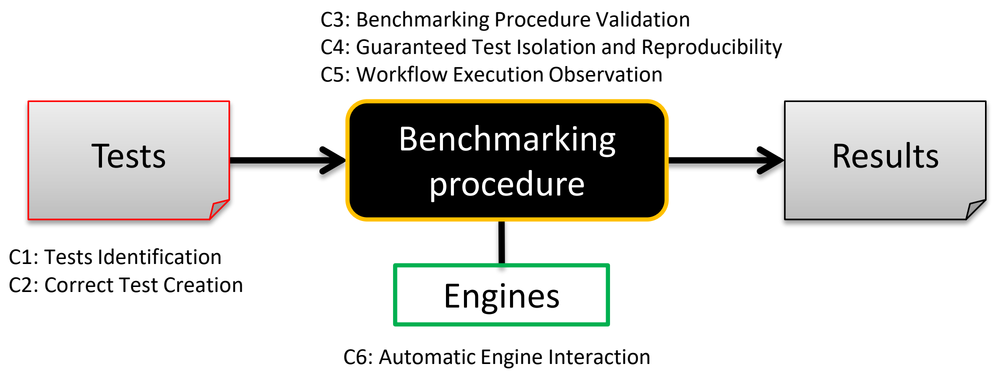

C3: Benchmarking Procedure Validation
C4: Guaranteed Test Isolation and Reproducibility
C5: Workflow Execution Observation
Tests
Benchmarking procedure
Results
C1: Tests Identification
C2: Correct Test Creation
Engines
C6: Automatic Engine Interaction

Fig. 1: Big Picture of Workflow Engine Benchmarking

competing solutions to the challenges from Sect. 2. After that, we outline the types of relationships among the singular patterns in Sect. 5. This is followed by a discussion of related pattern languages in Sect. 6. Finally, the paper is concluded with a summary and an outlook on future work in Sect. 7.

2. CHALLENGES IN WORKFLOW ENGINE BENCHMARKING

In the following, we clarify the reoccurring challenges one faces when building a conformance or performance benchmark for workflow engines. These challenges are the crucial sources and motivation for gathering workflow engine benchmarking patterns. We then introduce Betsy and BenchFlow, the main source of knowledge and experience for the proposed patterns.

2.1 Big Picture of Workflow Engine Benchmarking

Workflows and workflow engines, or more abstract, process-aware information systems [van der Aalst 2013], are commonly used in the service-oriented computing domain to orchestrate services [Peltz 2003]. In short, a workflow is the machine-readable and executable representation of a business process in whole or part and a workflow engine is the software runtime environment that manages and controls the execution of workflow instances [WfMC 1995]. Today, two language standards are predominantly used for workflow specification and execution. These are BPEL [OASIS 2007] and BPMN [ISO/IEC 2013].

Benchmarks are an important tool in computer science that is needed to compare and analyze the quality provided by software systems [Huppler 2009]. Many aspects of software can be benchmarked, but often the focus resides on performance-related aspects, such as latency or throughput. When it comes to workflow engines, two major aspects have been in the spotlight. As indicated above, one of these is performance [Ferme et al. 2015; Bianculli et al. 2010b; Rosinosky et al. 2016; Daniel et al. 2011]. The second aspect is standard conformance, which reflects the fact that workflow engines are often standards-based products [Harrer et al. 2012; Geiger et al. 2015]. Benchmarking approaches for both aspects exist for both languages mentioned in the previous paragraph, for BPEL [Bianculli et al. 2010b; Harrer et al. 2012] and BPMN [Ferme et al. 2015; Geiger et al. 2015].

Figure 1 offers a big picture of workflow engine conformance and performance benchmarking, highlighting its four main elements: *tests*, the *engines* to be tested, the *benchmarking procedure*, and the benchmark *results*.

When a benchmark is conducted, *tests* are used to specify requirements, workloads, expectations or desired behavior of the engines under test. Next to the tests, a set of *engines* is the second input to the benchmark. They are the objects of study (i.e., the systems under test) that are to be evaluated so that they can be compared in a fair way. The *benchmarking procedure* is the tool for evaluating engines according to the tests and produces the corresponding benchmark *results*. These results should be constructed in an easily comprehensible fashion to allow for a straightforward interpretation. In the simplest case, they provide a ranking between the engines, supported by empirical evidence obtained through measurements and a calculation of key performance indicators.

2.2 Challenges

During workflow engine benchmarking, several challenges arise related to each element listed in Sect. 2.1. These challenges are non-trivial and correspond to reoccurring problems that need to be solved for every benchmark. Hence, they are the problems for which we propose patterns as a solution here. In total, we identified six challenges, numbered from C1 to C6, which we present in the following.

Regarding the tests, the major issues are about *Tests Identification (C1)* and *Correct Test Creation (C2)*. The tests should be suitable and representative of realistic usage scenarios. If this is not the case, the results produced by the benchmark are of no use. Since realistic tests can be non-trivial, it is important to ensure that they are free of issues, since even minor issues could have a considerable impact on the benchmark results.

Major challenges regarding the benchmarking procedure are *Benchmarking Procedure Validation (C3)*, *Guaranteed Test Isolation and Reproducibility (C4)*, and *Workflow Execution Observation (C5)*. As for the tests, quality assurance needs to be in place to make sure that there are no errors in the benchmarking procedure that might have an impact on the benchmark results. Since realistic test sets might be large, it is important to make sure that tests can be executed independently, regardless of the execution order, and regardless of whether execution takes place sequentially or in parallel. Moreover, as outlined by Kistowski et al. [v. Kistowski et al. 2015], reproducibility has to be ensured. Finally, a mechanism needs to be in place that helps to identify how and if the benchmark and singular tests are progressing. Since a benchmark might push an engine to its limits, it can easily be the case that an engine fails to make progress during execution, which needs to be detected and acted upon.

Regarding the engines, the major issue is *Automatic Engine Interaction (C6)*. The sixth challenge concerns the ability of the engine to participate in a benchmark in the first place. Test execution requires the evaluation of assertions or observation of behavior, so it is necessary that the engine has facilities in place that allow to communicate its state to the outside. Moreover, the engine needs to be properly installed and configured at the start of the benchmark so that it can operate correctly, otherwise meaningful results are unlikely.

2.3 Workflow Engine Benchmarking with BenchFlow and Betsy

We derived the patterns presented here from our long-standing work on workflow engine benchmarking in the context of two benchmarking systems or projects. Firstly, this is the *BPEL/BPMN Engine Test System* (Betsy)[2], which implements a conformance benchmark for workflow engines. Secondly, this is *BenchFlow*[3], which implements a performance benchmark in this area. Although these two systems are the initial motivators for developing this pattern language, there are more uses of the patterns in other systems as well. These uses are listed in the respective parts of the pattern descriptions.

Betsy has been introduced as a conformance evaluation tool for engines supporting BPEL [OASIS 2007] in 2012 [Harrer et al. 2012]. The initial aim of the tool was to judge the maturity of the standard support for BPEL in the industry, i.e., to see how well the standard is implemented by workflow engines in practice. Subsequently, the tool was extended with support for more and more BPEL engines [Harrer et al. 2013]. Eventually, we added support for benchmarking engines for BPMN [ISO/IEC 2013] in 2015 [Geiger et al.

[2]https://github.com/uniba-dsg/betsy
[3]https://github.com/benchflow/benchflow

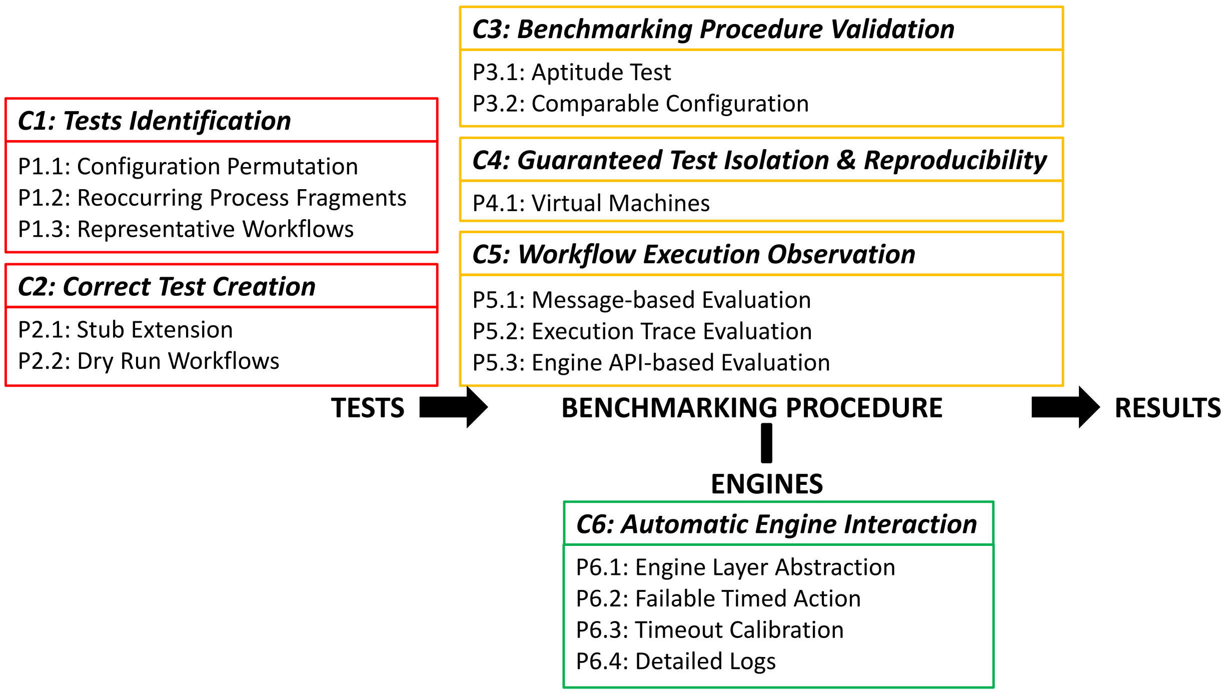

C3: Benchmarking Procedure Validation
P3.1: Aptitude Test
P3.2: Comparable Configuration
C1: Tests Identification
P1.1: Configuration Permutation
P1.2: Reoccurring Process Fragments
P1.3: Representative Workflows
C4: Guaranteed Test Isolation & Reproducibility
P4.1: Virtual Machines
C5: Workflow Execution Observation
P5.1: Message-based Evaluation
P5.2: Execution Trace Evaluation
P5.3: Engine API-based Evaluation
C2: Correct Test Creation
P2.1: Stub Extension
P2.2: Dry Run Workflows
TESTS
BENCHMARKING PROCEDURE
RESULTS
ENGINES
C6: Automatic Engine Interaction
P6.1: Engine Layer Abstraction
P6.2: Failable Timed Action
P6.3: Timeout Calibration
P6.4: Detailed Logs

Fig. 2: A Pattern Language For Engine Conformance and Performance Benchmarking

2015]. During this evolution, we tested many different approaches for tackling common problems that we faced. The multi-language support forced us to find reusable solutions that work for multiple engines which build on very different paradigms. Some of the solutions that we found useful during this work are now described in the form of patterns in this paper.

BenchFlow[4] has been developed as an end-to-end framework to simplify and automate reliable performance benchmarking of BPMN engines [Ferme et al. 2015; Skouradaki et al. 2016; Ferme et al. 2016a]. It reuses and integrates state of the art technologies, such as Docker[5], Faban[6], and Apache Spark[7] to reliably execute performance tests, automatically collect performance data, and compute performance metrics and statistics, as well as to validate the reliability of the obtained results.

3. WORKFLOW ENGINE BENCHMARKING PATTERN LANGUAGE

The pattern language for conformance and performance benchmarking we propose covers all the elements of a benchmark from Fig. 1, namely tests, benchmarking procedure, engines, and results, and is based on the challenges described in Sect. 2.2.

We organize the patterns we include in the language per challenge, as presented in Fig. 2, where the mapping to the specific benchmarking element is highlighted with the same set of colors used in Fig. 1. Patterns to identify the tests (C1) and correctly create tests (C2) are related to the test element of the benchmark, and concern the identification and quality assurance of test cases for a benchmark. The patterns

[4]https://github.com/benchflow
[5]http://docker.com
[6]http://faban.org
[7]http://spark.apache.org

Configuration Permutation (P1.1), Reoccurring Process Fragments (P1.2), and Representative Workflows (P1.3) help to identify the tests (C1). To correctly create tests (C2), the relevant patterns are Stub Extension (P2.1) and Dry Run Workflows (P2.2).

Patterns that help to validate the benchmarking procedure (C3), guarantee test isolation and reproducibility (C4), and observe the workflow execution (C5) concern benchmarking procedure guidelines for enabling comparability of results, and automating the benchmark environment. The patterns related to validating the benchmarking procedure (C3) are Aptitude Test (P3.1), and Comparable Configuration (P3.2). The one related to guaranteeing test isolation and reproducibility (C4) is Virtual Machines (P4.1). Patterns related to observing the workflow (C5) are Message-based Evaluation (P5.1), Execution Trace Evaluation (P5.2), and Engine API-based Evaluation (P5.3).

Engine related patterns, which deal with the challenges related to automating the interaction with the engines (C6), describe ways to instrument workflow engines for using them in a benchmark. These patterns are Engine Layer Abstraction (P6.1), Failable Timed Action (P6.2), Timeout Calibration (P6.3), and Detailed Logs (P6.4).

4. WORKFLOW ENGINE CONFORMANCE AND PERFORMANCE BENCHMARKING PATTERNS

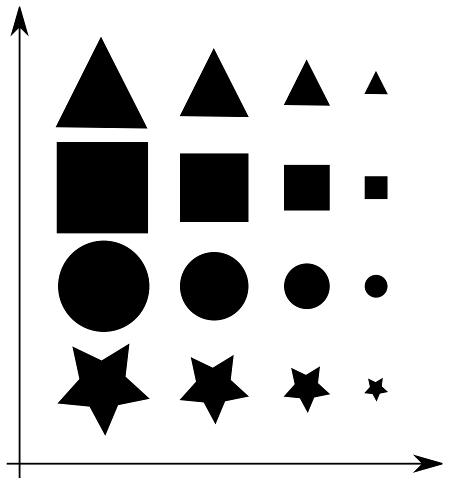
In this section, we present the workflow engine conformance and performance benchmarking patterns, mirroring the mapping of patterns to challenges introduced in Sect. 3.

For each pattern, we provide a unique *name*, a *summary*, the *context* and a list of the *problems* it addresses, which correspond to the challenges from Sect. 2.2. Furthermore, we describe the *forces*, the *solution* to said problems, outlining what the pattern does in an abstract form, and the *consequences* of applying the pattern. Lastly, we outline the *known uses* of a pattern and the *relations* it has to other patterns of our proposed pattern language.

4.1 Tests Identification (C1) Patterns

To *identify the tests (C1)*, one can determine the constructs of a modeling and/or execution language and apply Configuration Permutation *(P1.1)*. Alternatively, if a process model collection is available, one can use Reoccurring Process Fragments *(P1.2)* or Representative Workflows *(P1.3)* to identify the most frequently used process fragments and workflows within a given collection.

Configuration Permutation (P1.1)



Summary

Determine all variants, in which a construct of a workflow language (e.g., a language construct for conditional branching, such as an *exclusive gateway*[8] in BPMN) can be used, and cover them with test cases. The pattern captures how such test cases are derived from the defined structure of language constructs.

Context

Workflow modeling language specifications such as BPMN or BPEL contain a variety of language constructs including control-flow constructs such as conditionals and loops as well as data-flow constructs such as sending and receiving events. Language constructs may have many configuration options that modify their execution behavior. These configuration options are often not independent and result in different behavior when they are combined. In conformance benchmarking it is desirable to determine whether the configuration options of these language constructs are supported by the workflow engines implementing the specification. In this case, benchmarking means that the workflow engines are compared against standardized specifications and it is determined how well they conform to those specifications. In performance benchmarking this pattern ensures that all the possible variants of a construct, common or not, are tested.

Problem

How to include tests to cover all the possible variants in which a construct of a workflow language can be used?

Forces

- The main goal is to achieve completeness in the tests for a language construct, BUT without a systematic approach it is easy to forget language construct variants, although not every variant is used in practical scenarios. When striving for completeness, the problem of test explosion arises. Hence, the effort in regard to test creation and execution has to be kept under control.

Solution

(1) Identify a construct in the specification of a workflow language;
(2) Determine all configuration parameters and their range of values;
(3) Permutate them to get all configurations for the construct;
(4) Select the ones out of all configurations that are allowed by the language specification.

[8] An exclusive gateway, or XOR gateway, represents a decision in a workflow in which exactly one branch has to be taken.

Every allowed configuration of a construct may have multiple variants, such as boundary values or invalid values, resulting in another round of permutations of (1-4). Each variant is a test.

Example – In BPMN, there is the *exclusive gateway* construct which can be configured in three ways (the range of values is *standard*, *default*, and *mixed*): (i) *standard* with all outgoing sequence flows having conditions, (ii) an exclusive gateway with a sequence flow without a condition and marked as *default*, and (iii) one as a *mixed* gateway with both branching and merging capabilities. In this example, all three configurations are allowed in the specification and each configuration has exactly one variant resulting in three tests in total for that language construct.

Consequences

Benefits – The consequence of using this pattern is a more comprehensive set of tests by deriving tests in a systematic fashion. With these tests it can be determined which variants of the constructs are supported on a workflow engine. This knowledge can drive the creation of REPRESENTATIVE WORKFLOWS (P1.3) that can be executed on the majority of today's workflow engines.

Liabilities – The strive for completeness advocated by this pattern complicates the subsequent execution of the benchmark, since execution time and resource consumption increases with the number of tests (i.e., combinatorial explosion). This can be mitigated by reducing the test execution time with VIRTUAL MACHINES (P4.1). The more tests that have to be created, the higher is the effort required to create correct tests. This effort can be reduced through STUB EXTENSION (P2.1) whereas DRY RUN WORKFLOWS (P2.2) ensure that the tests are without faults.

Known Uses

(1) From a more general point of view, the principle underlying this pattern is known as *combinatorial design testing* [Cohen et al. 1997].
(2) The pattern has been used for BPEL conformance benchmarking [Harrer et al. 2012].
(3) It has also been used for BPMN conformance benchmarking [Geiger et al. 2015].

Relations

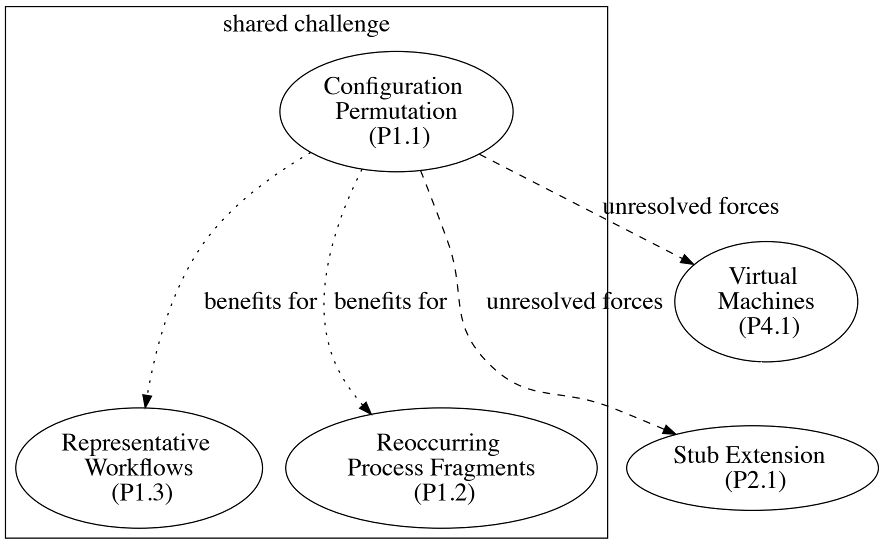
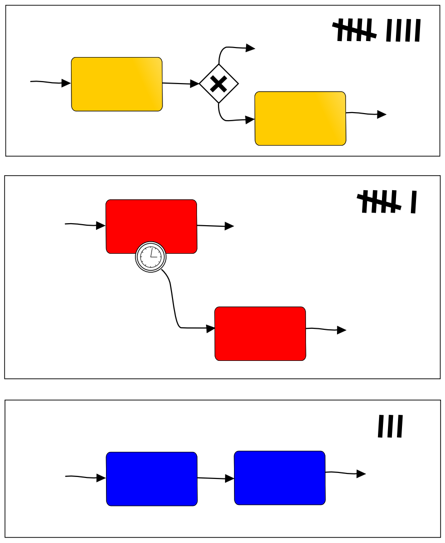


Fig. 3: Relations of P1.1 with other Patterns

Shared Challenge – If data on the occurrence frequency of language constructs or a collection of representative workflows is available, then Reoccurring Process Fragments (P1.2) and Representative Workflows (P1.3) are viable alternatives to solve the shared challenge of those three patterns.
Unresolved Forces – Execution time can be kept under control by introducing Virtual Machines (P4.1). Test creation effort can be reduced through Stub Extension (P2.1).
Benefits for – Reoccurring Process Fragments (P1.2) and Representative Workflows (P1.3) benefit from Configuration Permutation (P1.1).

Reoccurring Process Fragments (P1.2)



Summary

Identify the most important or most relevant combinations of workflow language constructs (i.e., process fragments) as test cases based on their occurrence in real-world process collections.

Context

Workflow languages contain many different language constructs. In a workflow model, configurations of these constructs are combined to implement a desired piece of functionality. All combinations of constructs that are permitted according to the specification of the language should behave correctly, especially those that are frequently needed in practice, because we can expect the workflow engines to execute them often. This pattern applies to conformance benchmarking, since all combinations of language constructs should behave correctly, and also to performance benchmarking, since combining language constructs should not yield unexpected performance pitfalls.

Problem

How to identify common control-flow structures used within different workflows as test cases?

Forces

- In workflow engine benchmarking it is desirable to test all ways in which basic language constructs can be combined with each other, BUT this potentially leads to a combinatorial explosion of the amount of test cases.
- More important combinations of language constructs should be prioritized, BUT this requires information about which constructs or combinations of constructs are used more frequently, because otherwise users of the benchmark will struggle to see its relevance.
- Focusing on important combinations of language constructs can increase the relevance of a benchmark, BUT the importance of a construct might depend on an application domain. Therefore, applying this pattern can turn a benchmark more relevant to one domain, but less relevant to others.

Solution

(1) Gather a large corpus of workflows;
(2) Identify the the most common fragments in these workflows by counting their occurrence frequency;
(3) Create tests based on the most important (i.e., reoccurring) fragments.

Example – Well-known reoccurring process fragments in the area of workflows are the so-called workflow patterns [van der Aalst et al. 2003; Wohed et al. 2006]. These patterns capture control-flow structures that the authors of the above mentioned studies have found to be common in workflow models by analyzing the capabilities of a wide range of engines. A concrete example is the splitting of the control-flow into separate branches, where on execution one of the branches is chosen exclusively based on a condition (called exclusive choice pattern). Listing 1 outlines the code of the test case for this pattern for benchmarking BPEL engines with Betsy.

Listing 1: Code outline of the Betsy test for the exclusive choice workflow pattern in BPEL

```
<process>
   <partnerLinks />
   <variables />
   <sequence>
        <!--Consumes test input parameters-->
        <receive />
        <!--Takes alternative control-flow paths based on input parameters-->
        <if>
          <assign />
        </if>
        <else>
          <assign />
        </else>
        <!--Returns a different value depending on the path taken that is used for
            checking test success-->
        <reply />
   </sequence>
</process>
```

Consequences

Benefits – The pattern intends to increase the meaningfulness of a benchmark by focusing on the most frequently used aspects of a workflow language. This focus helps to avoid a combinatorial explosion of the amount of test cases that would occur when trying to cover all combinations of constructs. Since occurrence frequencies of language constructs can be sensitive to the considered process collection, the outcome of applying this pattern is also highly domain-dependent. A strong tie to a certain domain can be both, a benefit or a liability, for a benchmark. This depends on whether the benchmark is intended as a general-purpose or a domain specific evaluation.

Liabilities – The application of the pattern does not guarantee complete coverage with regards to a process modeling language specification. If this is a desired property for the benchmark, then Configuration Permutation (P1.1) should be applied as well. The effort of creating test cases based on reoccurring process fragments can be mitigated by applying Stub Extension (P2.1). Furthermore, Dry Run Workflows (P2.2) can be applied to validate the correctness of created test cases. Moreover, the pattern is meant to test the workflow engine in handling frequent structures, but does not necessarily cover business critical processes, that might contain different constructs and structures. For covering the last situation, Representative Workflows (P1.3) can be used to select representative workflow with different structural characteristics, even if they are not so frequent.

Known Uses

(1) In the performance testing literature, the principle underlying this pattern is used for obtaining a *kernel workload* [Dujmović 2010].

(2) This pattern has been used by Bianculli et al. in SOABench [Bianculli et al. 2010a] to define performance tests for BPEL engines.
(3) It has been used for BPEL conformance benchmarking with Betsy on the basis of workflow patterns [Harrer et al. 2013].
(4) The pattern has also been applied for BPMN conformance benchmarking with Betsy using workflow patterns [Geiger et al. 2015].
(5) Finally, micro-benchmarks for performance with BenchFlow were implemented using workflow patterns [Skouradaki et al. 2016].

Relations

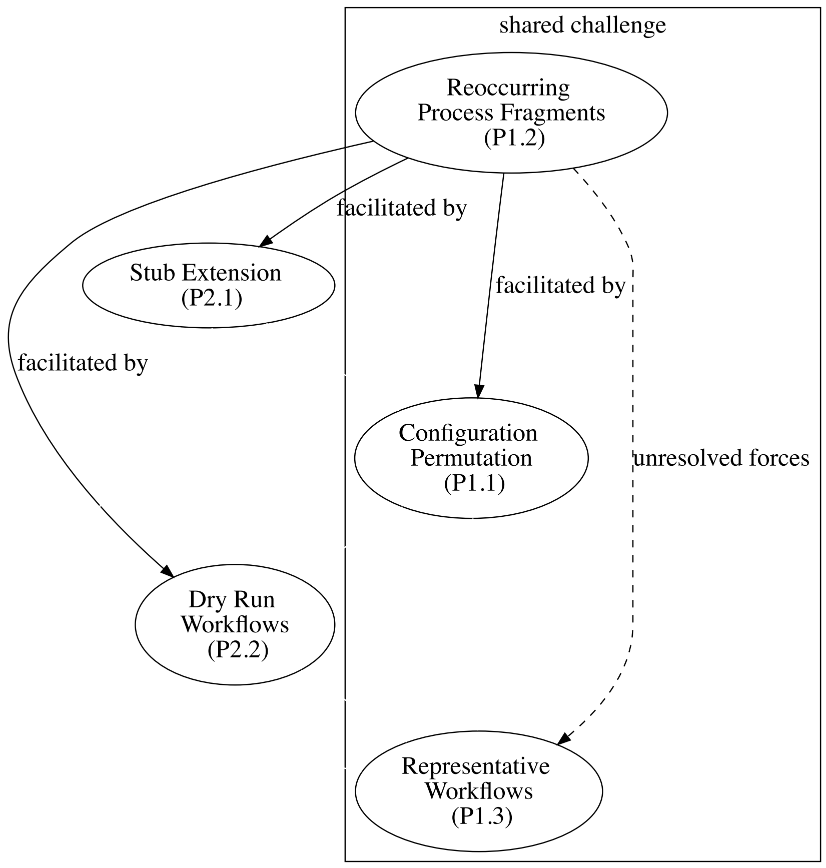
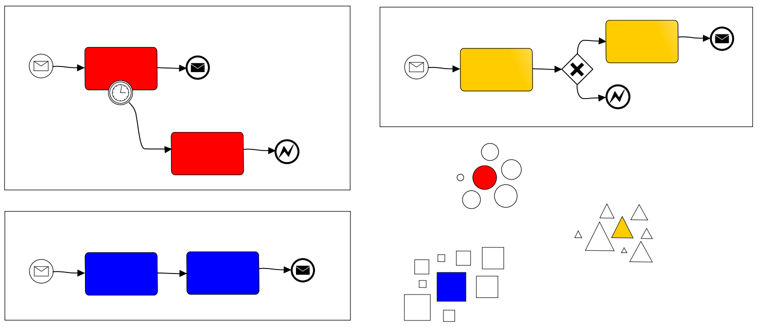


Fig. 4: Relations of P1.2 with other Patterns

Shared Challenge – The pattern shares the challenge it addresses, namely how to identify the test cases for the benchmark, with the patterns REPRESENTATIVE WORKFLOWS (P1.3) and CONFIGURATION PERMUTATION (P1.1).
Facilitated by – The creation of test cases using this pattern can be supported by applying STUB EXTENSION (P2.1). CONFIGURATION PERMUTATION (P1.1) helps to see if certain configurations of constructs are

not supported even when used in isolation. If this is the case, there is no point in retesting the same configurations as part of a larger test case created using this pattern. Furthermore, Dry Run Workflows (P2.2) support correctness checking and validation of test cases that were created using this pattern.
Unresolved Forces – Representative Workflows (P1.3) can be used to select representative workflows with different structural characteristics, even if they are not so frequent.

Representative Workflows (P1.3)



Summary
Pick a small set of representative workflows out of a large collection based on their static/structural properties.

Context
Workflow engines are designed to host many different workflow models at the same time [WfMC 1995]. These collections of models can thus be very large with diverse models that are executed to achieve business goals. This pattern suggest to select the most representative models from an overall collection. It applies mainly to performance benchmarking.

Problem
How to select representative workflows to be used for benchmarking, out of a large model collection, so that the benchmark can be executed in a reasonable amount of time?

Forces
- Real-world collections of workflow models can be very large, BUT benchmarks require time to be executed [Ferme et al. 2016b]. Thus, it might not be feasible to execute all the models from a collection.
- It is possible to select a subset of workflows from a collection for using them in a benchmark, BUT the workflows included in the benchmark should be representative of the overall collection.

Solution
(1) Gather a large corpus of workflows;
(2) Apply clustering techniques [Jain and Dubes 1988] on a process collection, using workflows' static and structural complexity metrics [Lassen and van der Aalst 2009; Vanderfeesten et al. 2008] as features;
(3) Select the clustroid [Ganti et al. 1999] workflow of each cluster as representative of the collection and include them in the benchmark;
(4) Execute the benchmark using the process models obtained out of step 3.

Consequences
Benefits – The subset of selected workflows reduces the execution time of the benchmark in comparison to time required for the full corpus of workflows. The subset also contains representative workflows that are rare within the corpus because they are also represented through their own clusters.

Liabilities – Depending on the variability of the model features, the analysis may fail to reveal a small number of clusters. Moreover the obtained sets of process models might not be representative for the full workflow corpus, and thus not be able to evaluate the performance of the workflow engines realistically.

Known Uses

(1) In the performance benchmarking literature, a similar technique is applied to define kernel workloads [Dujmović 2010], e.g., workload representing the most common usages of a language constructs, as for example the LINPACK's benchmark workload [Dongarra and Luszczek 2011].

(2) The technique is used in the context of BenchFlow [Ivanchikj 2014] to derive representative workflows for performance benchmarking.

(3) Argenti [Argenti 2015] applied this technique to derive workflows representative of different clusters to execute performance benchmarks simulating differently sized companies.

Relations

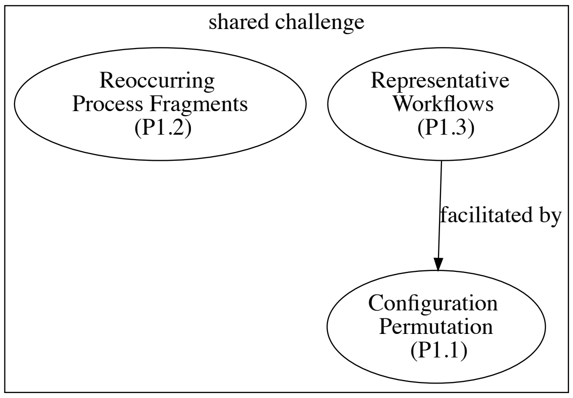

shared challenge
Reoccurring Process Fragments (P1.2)
Representative Workflows (P1.3)
facilitated by
Configuration Permutation (P1.1)

Fig. 5: Relations of P1.3 with other Patterns

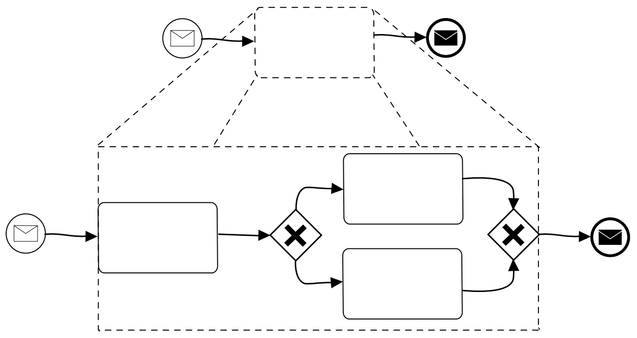
Shared Challenge – The pattern shares the challenge it addresses, namely how to identify the test cases for the benchmark, with the patterns Reoccurring Process Fragments (P1.2) and Configuration Permutation (P1.1).

Facilitated by – Representative Workflows (P1.3) benefits from the application of Configuration Permutation (P1.1), because knowing which language features are supported by the workflow engines to be benchmarked helps to select workflows containing language features that can actually be executed as part of the benchmark.

4.2 Correct Test Creation (C2) Patterns

To *correctly create tests (C2)*, one can derive tests using Stub Extension *(P2.1)*, which ensures a certain degree of correctness by design. Moreover, before starting the actual benchmark, Dry Run Workflows *(P2.2)* verify that the created tests can actually be correctly executed by the engines.

Stub Extension (P2.1)



Summary

Increase efficiency and ensure correctness by creating tests through the extension of the same skeleton of a workflow model.

Context

Benchmarks can require the creation of a huge set of test cases to adequately cover their objectives. Even more important, all test cases need to be correct and, due to the amount of tests that need to be created, support for test correctness by design, as offered by this pattern, is helpful. The pattern is applicable in the context of conformance and performance benchmarking.

Problem

How to ensure the tests are created correctly?

Forces

- Workflow modeling languages can express arbitrarily complex processes with a large number of modeling constructs, BUT usually there is a minimal number of language constructs that are always needed to turn a workflow into an executable piece of software.
- Different test cases of a benchmark should focus on different aspects of a language, BUT they all require the presence of the most foundational constructs, which will need to be reused in many workflows.
- Benchmarks often consist of a huge set of test cases, BUT a lack of consistency when creating the test cases can easily lead to errors.
- Building every test case from scratch separately is feasible, BUT error-prone.

Solution

(1) Determine the most basic features that are required to build an executable workflow, a workflow stub;
(2) Mark extension points in the stub, where further language constructs can be inserted;
(3) Start the implementation of test cases with a copy of the stub and build upon its extension points.

A workflow stub is a very basic workflow, which is extended for all tests, so that the extension contains solely the feature under test. The stub itself provides extension points, where the feature under test can be put. The rest is the minimal overhead required to observe the feature under test. This way, all tests follow the same structure, and when looking at the difference between the test and the stub, the feature under test can be easily identified. In essence, the principle underlying this pattern is a general test design principle: tests should be as concise and minimal as possible to make sure that they focus on the aspects relevant to the test only.

Example – The stub used for BPEL conformance benchmarking in Betsy consists of a *Receive* activity (to start a new workflow instance) and a *Reply* activity (to observe correct termination of the instance), contained in a *Sequence* activity (to order the flow of control). The main extension point is between the *Receive* and the *Reply* activity. An outline of the code of this stub is depicted in Listing 2.

Listing 2: Stub for BPEL Workflows

```
<process>
   <partnerLinks />
   <variables />
   <sequence>
        <receive />
        <!-- Test implementation-->
        <assign />
        <reply />
        <!--More test implementation, if message exchanges are involved-->
   </sequence>
</process>
```

Consequences

Benefits – The application of the pattern results in a standardization of the structure of test cases, which can benefit understandability. The effort of creating new test cases is reduced, since a starting template for a test case is available. This is beneficial if a huge set of test cases needs to be created. By using the existing template, there is less potential for accidentally introducing errors during test creation.

Liabilities – Although the pattern supports the efficient creation of correct tests, it may not always be applicable or desirable to have a standardized structure of test cases. The reason for this is that the structure of a stub may contradict a particular type of practical use case or reoccurring process fragment.

Known Uses

(1) Stubs are frequently used in testing workflows, but rather for the purpose of replacing external communication parties, e.g. [Li et al. 2005].
(2) This pattern was applied in Betsy for BPEL conformance benchmarking [Harrer et al. 2012]
(3) Moreover, it has been applied in BPMN conformance benchmarking [Geiger et al. 2015].
(4) Workflow stubs have also been used for computing the distance between workflow models for the purpose of pattern support assessment [Lenhard et al. 2011].

Relations

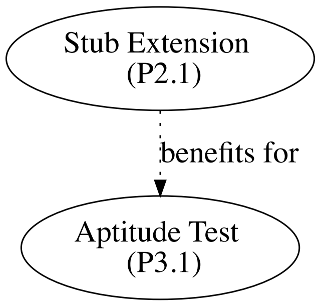
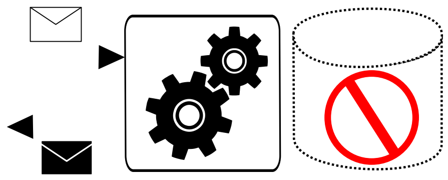


Fig. 6: Relations of P2.1 with other Patterns

Benefits for – If the stub is fully functional, it can act as an APTITUDE TEST (P3.1).

Dry Run Workflows (P2.2)



Summary

Execute the benchmark workflows in isolation using in-memory engines to test the correctness of workflow execution.

Context

Benchmarks can contain many different workflows that can possibly fail at runtime due to dynamic errors. This pattern helps in reducing the number of runtime errors, during the execution of the benchmark.

Problem

How to efficiently ensure that the workflows can be correctly executed on all the engines that are part of the benchmark before running the actual benchmark?

Forces

- Not all engines fully support all workflow modeling language features, in all the possible configuration variants. Applying statics analysis on the workflows part of the benchmark ensures that the workflows are syntactically and semantically valid, BUT in order to check that the targeted engines are able to execute such workflows they have to be deployed on the engines to be executed, and the execution outcome has to be verified.

Solution

(1) Deploy a workflow on an in-memory engine instance;
(2) Execute at least one workflow instance for each workflow part of the benchmark;
(3) Verify that the execution state of the workflow instance(s) is as expected.

For improved performance of the dry run, the engines can be deployed using a non-persistent configuration.

Consequences

Benefits – Failed dry run executions allow to detect dynamic errors early, avoiding to run the entire benchmark.

Liabilities – Some failures during the real benchmark execution can still occur due to other factors, e.g., engine overload.

Known Uses

(1) In the field of software testing, the principle underlying this pattern is well-known as *dry run testing* [Schmidt 2013]. Originally, the term *dry run* comes from firemen practicing for an emergency and executing their job without water, hence *dry*.
(2) In software performance engineering, it is known as *smoke test* [Molyneaux 2014].
(3) BenchFlow relies on this technique to validate the correct dynamic behavior of the workflow instances executed as part of the benchmark, for all the engines that take part in it.

Relations

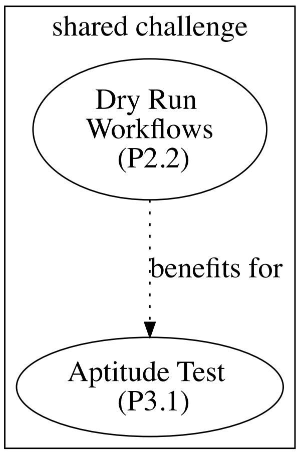


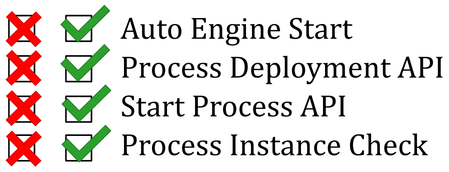
Fig. 7: Relations of P2.2 with other Patterns

Shared Challenge – Dry Run Workflows (P2.2) can be used to conduct the Aptitude Test (P3.1).
Benefits for – The pattern can be used to execute an Aptitude Test (P3.1).

4.3 Benchmarking Procedure Validation (C3) Patterns

To *validate the benchmarking procedure (C3)*, one can apply Aptitude Test *(P3.1)* and Comparable Configuration *(P3.2)*.

Aptitude Test (P3.1)



Summary
The Aptitude Test (P3.1) describes the minimal requirements for an engine for participating in a benchmark and automates the checking of these requirements.

Context
Especially when integrating a new engine or a new version of it into a benchmark, it needs to be made sure that the engine can correctly participate in it. Also when updating the benchmarking procedure, it needs to be made sure that the interaction with the engines is still correct. The pattern is relevant to conformance and performance benchmarking alike.

Problem
Are there enough engines available which fulfill specific requirements so that a benchmark is worthwhile?

Forces

- The motivation for an Aptitude Test (P3.1) is to find out if a benchmark is worth doing. The test avoids a waste of resources in case a benchmark is meaningless, BUT the test could be considered an overhead in case of a positive result.
- The Aptitude Test (P3.1) can be conducted through Dry Run Workflows (P2.2) to determine the aptitude of the complete system more quickly, BUT this can result in different outcomes as the Aptitude Test (P3.1) is not performed in the actual benchmarking environment.

Solution

(1) Determine minimal requirements for an engine to participate in the benchmark;
(2) Define an aptitude test that automatically determines whether the minimal requirements are met;
(3) Abort the complete benchmark if an engine does not pass the aptitude test.

Example – In conformance benchmarking, the Aptitude Test (P3.1) checks whether 1) an engine can be installed and started automatically, 2) a minimal workflow can be deployed automatically to that engine, and 3) that the engine can execute the deployed minimal workflow correctly. Additionally, for performance benchmarking, 4) the engine should log when each process instance was started and finished.

Consequences

Benefits – An Aptitude Test (P3.1) results in the early termination of a benchmark in case the benchmarking procedure is faulty, or the execution environment of the benchmark does not work as expected. Especially during phases of benchmark development, this pattern results in immediate feedback on errors that renders the benchmark useless, and which otherwise would only have become obvious after the completion of a full benchmark.

Liabilities – The application of the pattern causes an overhead of an additional test per performed benchmark. Typically, this overhead is acceptable. In case the overhead of the additional test is not acceptable, it is possible to perform the Aptitude Test (P3.1) once for a group of benchmarks.

Known Uses

(1) In a more general sense, aptitude tests are used during job interviews to evaluate verbal or reasoning skills of applicants as a baseline for their employability [Hartigan and Wigdor 1989]. This principle is adopted in this pattern to the workflow engine benchmarking context.
(2) Betsy uses a BPEL-specific aptitude test for BPEL conformance benchmarking, named *Sequence*, containing a receive-assign-reply triplet (see Message-based Evaluation (P5.1)).
(3) It also uses a BPMN-specific aptitude test for BPMN conformance benchmarking, named *SequenceFlow*, containing a start and end event, with corresponding script tasks to allow to observe the events (see Execution Trace Evaluation (P5.2)), connected through sequence flows.
(4) BenchFlow uses Aptitude Test (P3.1) as well to determine whether the engine can take part in the performance benchmarks [Ferme et al. 2016b]. It is combined with Dry Run Workflows (P2.2).

Relations

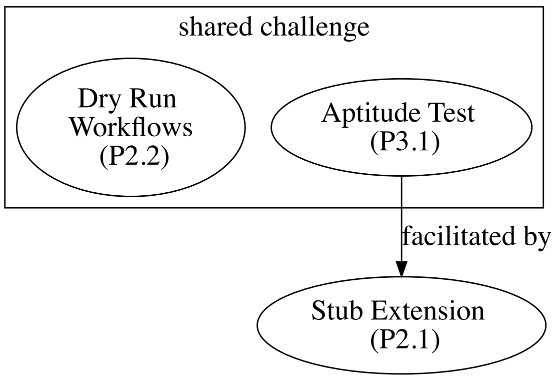
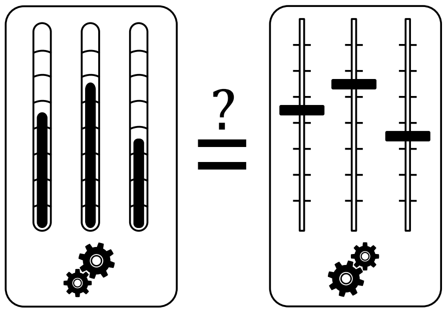


Fig. 8: Relations of P3.1 with other Patterns

Shared Challenge – The APTITUDE TEST (P3.1) can be conducted through DRY RUN WORKFLOWS (P2.2) to determine the aptitude more quickly.
Facilitated By – The APTITUDE TEST (P3.1) be used as a stub for STUB EXTENSION (P2.1) or vice-versa.

Comparable Configuration (P3.2)



Summary
The pattern has the goal of enabling comparability among the results of a benchmark involving more than one engine.

Context
This pattern is mainly relevant in performance benchmarking, but it also applies to conformance benchmarking when different alternative configurations impact the language constructs' configurations the engine can execute.

Problem
How to enable the possibility to compare results obtained by executing the same benchmark with different engines?

Forces

- Different engines might have different default value for similar configuration options, making it difficult to compare the benchmark results, BUT the goal of a benchmark is usually to compare different systems in a fair way.

Solution

(1) Determine the configuration options that can impact the performance or construct support of a workflow engine;
(2) Identify all identical or similar configuration options among the different workflow engines;
(3) Update the configuration option values accordingly so that all the engines use the same configuration, or at least one that is as similar as possible.

Example – In performance benchmarking, one has to ensure, e.g., that the logging level of the different engines is always the same, and that the application stack on top of which the engines rely on (e.g., application server) is set to the same version.

Consequences

Benefits – By setting the same configuration, comparability of results is enhanced.

Liabilities – It is sometimes difficult or impossible to set the same configuration, due to unavailability of certain options for some engines. In this case, the comparison of results must take into account the possible differences due to the different configuration.

Known Uses

(1) BenchFlow compares the performance of the engines while using the same configuration for all the engines [Skouradaki et al. 2016; Ferme et al. 2016a; Ferme et al. 2017].
(2) The pattern has also been used by Bianculli et al. [Bianculli et al. 2010a] to compare the performance of open-source and commercial BPEL workflow engines.
(3) In [Daniel et al. 2011] the authors make sure to use the same deployment stack for all engines, which are however tested in their default configuration.

4.4 Guaranteed Test Isolation and Reproducibility (C4) Patterns

Virtual Machines *(P4.1)* can be applied to *guarantee test isolation and reproducibility (C4)*.

Virtual Machines (P4.1)

Summary

The execution-ready state of an engine is captured in a snapshot of a dedicated virtual machine.

Context

Test isolation is a central property in a benchmark that is relevant for performance and conformance benchmarking alike.

Problem

How to guarantee test isolation and enhance reproducibility of the results?

Forces

- Every test in a benchmark needs to stand on its own, BUT execution runs of an engine may influence subsequent executions on the same engine instance. Thus, it is not permissible to reuse an instance for multiple tests.
- It would be more efficient to reuse the same engine for all tests, BUT this would lead to a coupling of test cases and failing to isolate tests properly renders a benchmark invalid.
- An fresh instance of an engine can be installed for every test case, BUT this can lead to a significant increase in the time taken and in the computing resources consumed by a benchmark.

Solution

(1) Create a virtual machine with a snapshot of a freshly installed and running engine upfront. This requires a one-time setup and effort for every engine that participates in the benchmark;
(2) With such a snapshot in place, each test can be executed in isolation: the snapshot can easily be restored before each test and be discarded afterwards, resulting in test isolation with a low temporal overhead.

Example – In Betsy, a reinstallation of the engine for every test was not possible for a number of engines, in particular for commercial ones, because the installation routine took several hours to complete [Harrer et al. 2014b]. Therefore, the operating system state of was captured in a virtual machine snapshot after a fresh installation of these engines. The benchmarking procedure of betsy was then modified to start up the virtual machine snapshot, deploy test cases to the snapshot and evaluate test execution, and finally to reset the virtual machine state back to the original snapshot for the next test case.

Consequences

Benefits – By using virtual machines, it is possible to guarantee test isolation with a less significant execution time overhead than by reinstallation.

Liabilities – For virtual machines, there is typically a substantial RAM, disk and performance overhead [Barik et al. 2016].

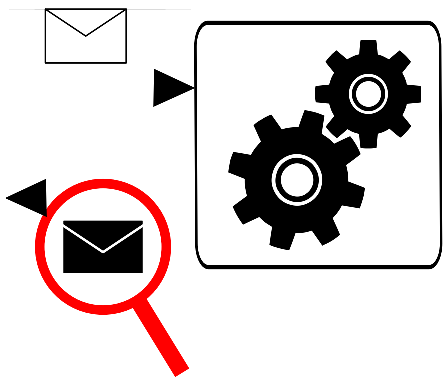
Known Uses

(1) Daniel et al. [Daniel et al. 2011] rely on virtual machines for performance benchmarking of engines.
(2) Bianculli et al. [Bianculli et al. 2010a] rely on virtual machines for performance benchmarking of engines.
(3) Rosinosky et al. [Rosinosky et al. 2016] use virtual machines provided on Amazon Web Services public cloud for benchmarking the performance of a workflow engine.
(4) Support for virtual machines has been implemented in Betsy, primarily for BPEL conformance benchmarking [Harrer et al. 2014b].

4.5 Workflow Execution Observation (C5) Patterns

To *observe the workflow execution (C5)*, Message-based Evaluation *(P5.1)*, Execution Trace Evaluation *(P5.2)* and Engine API-based Evaluation *(P5.3)* are applicable.

Message-based Evaluation (P5.1)



Summary

Direct communication with workflow instances in the form of message passing is used to evaluate the correctness of the execution of a test.

Context

The pattern advocates a live interaction with an engine during the execution of the benchmark. It is mainly relevant to conformance benchmarking since message passing might have an undesirable impact on the execution performance of a workflow instance. It can be applied in performance benchmarking as well, in case message passing features are the target of the benchmark.

Problem

How to observe the execution behaviour of a workflow during the execution of a test for evaluating its outcome?

Forces

- To verify that language features are implemented correctly, it is necessary to investigate how they are executed by the engine. BUT since engines are designed to hide the internal execution behavior of workflows, it is usually impossible to investigate the flow of execution directly (i.e., without relying on execution side-effects).
- Sending messages from and to a workflow instance allows to observe execution state, BUT it can have an impact on execution performance.

Solution

(1) Include functionality in the workflow models to consume and reply to messages from the outside. Use small communication interfaces with few methods and few message types, since this reduces the complexity of test cases;

(2) During test execution, let the benchmarking system send messages to a workflow instance, and store the reply;

(3) After the execution of the test case, evaluate the payload of the message with the expected result, to judge success or failure of the test.

Example – In BPEL conformance benchmarking, Betsy communicates with BPEL instances through six different types of SOAP messages that map to three different operations that can be supported by a test case.

Then, betsy evaluates the outcome of a test by checking the response messages. The messages exchanged are very simple, e.g., single string literals or numbers, to maintain a low impact on the overall complexity of a test. Listing 3 outlines the structure of the test interface that betsy uses, in particular the message types.

Listing 3: Outline of the WSDL code of the web service interface used by Betsy

```
<definitions name="TestInterface">
   <partnerLinkType />
   <property />
   <propertyAlias />

   <!-- Small set of simple message types-->
   <types>
        <xsd:schema>
            <xsd:element name="testElementSyncRequest" type="xsd:int"/>
            <xsd:element name="testElementSyncResponse" type="xsd:int"/>
            <xsd:element name="testElementSyncFault" type="xsd:int"/>
            ...
        </xsd:schema>

   <!-- Messages that reference the types-->
   <message />*

   <!-- A small set of operations that communicate the messages-->
   <portType>
        <operation name="startProcessSync">
            <input name="syncInput" message="tns:executeProcessSyncRequest"/>
            <output name="syncOutput" message="tns:executeProcessSyncResponse"/>
            <fault name="syncFault" message="tns:executeProcessSyncFault"/>
        </operation>
        ...
   </portType>

   <binding />
   <service />
</definitions>
```

Consequences

Benefits – Using the pattern, the execution behavior of workflow instances can be observed in a relatively direct and straight-forward way. It avoids the need for making use of native engine-specific APIs, and keeps the benchmarking infrastructure more independent from the particularities of different engines.

Liabilities – Applying this pattern influences the structure of the test cases, possibly increasing their complexity. Moreover, it comes at a cost of execution performance of workflow instances, which may not be acceptable in some cases.

Known Uses

(1) This form of specification-based testing is common in the area of service-oriented systems and web services [Bozkurt et al. 2012].
(2) Testing tools, such as SoapUI[9], use this strategy as well.
(3) In Betsy, the pattern is applied for BPEL conformance benchmarking.

[9]https://www.soapui.org/

(4) Bai et al. [2005] use this pattern to test web services.

Relations

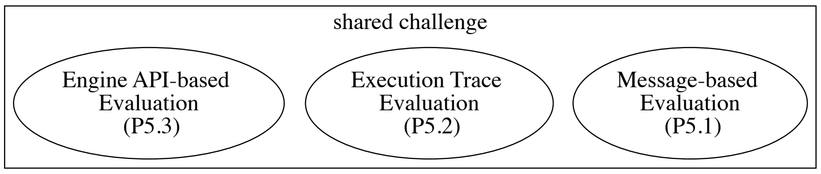
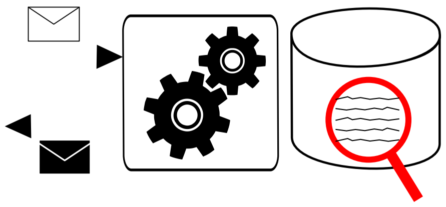

shared challenge
Engine API-based Evaluation (P5.3)
Execution Trace Evaluation (P5.2)
Message-based Evaluation (P5.1)

Fig. 9: Relations of P5.1 with other Patterns

Shared Challenge – EXECUTION TRACE EVALUATION (P5.2) and ENGINE API-BASED EVALUATION (P5.3) share the challenge of observing the workflow instance execution state to enable test evaluation with this pattern.

Execution Trace Evaluation (P5.2)



Summary

Workflow instances can be programmed to serialize their execution state at specific points in time. The sequences of such traces can be used to evaluate test execution.

Context

The pattern is mainly relevant for conformance benchmarking. As is the case for MESSAGE-BASED EVALUATION (P5.1), the problem is that the internal execution behavior of a workflow is not visible to the benchmarking system. Some form of output is needed from a workflow that can be evaluated by the benchmarking environment. The pattern tries to find a trade-off between the need for evaluating the outcome of tests in the benchmark and the execution performance of a test, which might be affected negatively.

Problem

How to observe the workflow execution behaviour?

Forces

- Observing the behavior of workflow instances through log traces allows asserting their execution correctness, BUT at a cost in execution performance.
- Using a small set of different standardized log traces results in smaller execution traces that are easier to comprehend, BUT may result in too generic traces that convey no semantics or meaning, resulting in assertions that are harder to comprehend.

Solution

(1) Define possible log traces;

(2) Configure the engine or the workflow to write log traces to disk;
(3) Execute the workflow;
(4) Optionally, inspect DETAILED LOGS (P6.4) and convert log statements to log traces;
(5) The benchmarking framework then reads the log traces after the execution of the test and compares them with expected ones.

Example – In BPMN conformance benchmarking, Betsy writes log traces through BPMN script tasks. Each script task writes a different log trace. By comparing the actual log traces with the expected log traces, it can be asserted if the execution was correct. Listing 4 outlines the BPMN code used for the test cases in Betsy. At decisive positions in the control-flow graph, a script task is inserted and the *script* element captures the log message to be written. When Betsy operationalizes the test case for a concrete engine, it picks up the message and inserts a script that the respective engine can execute.

Listing 4: Outline of the logging mechanism using BPMN script tasks in Betsy

```
<definitions>
   <process>
   <!-- Start of test case implementation-->
   <sequenceFlow id="SequenceFlow_1" sourceRef="start" targetRef="ScriptTask_1" />

   <scriptTask id="ScriptTask_1">
            <incoming>SequenceFlow_1</incoming>
            <outgoing>SequenceFlow_5</outgoing>
            <script>SCRIPT_task1</script>
   </scriptTask>

   <sequenceFlow id="SequenceFlow_5" sourceRef="ScriptTask_1" targetRef="
       ExclusiveGateway_1"/>

   <!-- Further test case implementation-->
   </process>
</definitions>
```

Consequences

Benefits – The correctness of the execution behavior of workflow instances can be observed. Assertions can be written in a concise fashion through a small number of standardized log traces.

Liabilities – Using this pattern reduces the performance of the system. That makes this pattern unsuitable for performance benchmarking. Moreover, test assertions might be hard to comprehend, if the log traces are too generic.

Known Uses

(1) This pattern is used frequently in the field of process mining [Rozinat and van der Aalst 2008].
(2) It is used in Betsy for BPMN conformance testing. For BPMN, MESSAGE-BASED EVALUATION (P5.1) is not applicable because of a lack of detailed support for sending and receiving messages. In script tasks, log traces are written to a log file. Moreover, engine specific logs are checked and additional log traces are created based on them. This is useful for conditions like the detection of whether a workflow did complete correctly.
(3) Execution traces are used for asserting whether workflow patterns [Russel et al. 2006] are supported by engines [Lenhard et al. 2011].

Relations

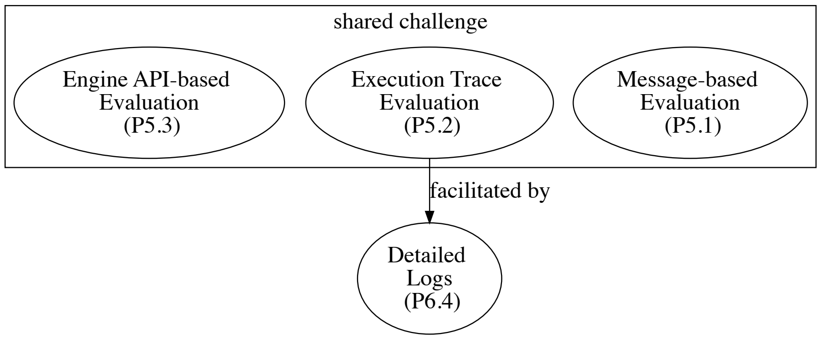
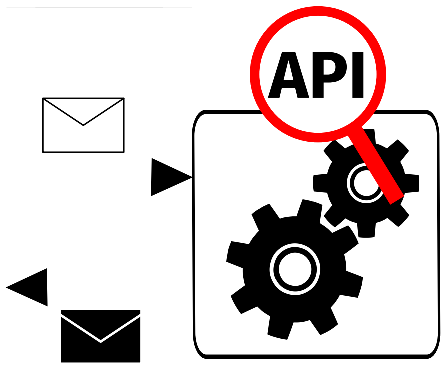

shared challenge
Engine API-based Evaluation (P5.3)
Execution Trace Evaluation (P5.2)
Message-based Evaluation (P5.1)
facilitated by
Detailed Logs (P6.4)

Fig. 10: Relations of P5.2 with other Patterns

Shared Challenge – MESSAGE-BASED EVALUATION (P5.1) and ENGINE API-BASED EVALUATION (P5.3) are alternatives to this pattern.
Facilitated by – DETAILED LOGS (P6.4) can be used to support an implementation of this pattern.

Engine API-based Evaluation (P5.3)



Summary
Engine APIs can be queried about the execution state of workflow instances to infer the success or failure of a test execution.

Context
This pattern is relevant to conformance and performance benchmarking. As is the case for MESSAGE-BASED EVALUATION (P5.1) and EXECUTION TRACE EVALUATION (P5.2), the problem is that the internal execution behavior of a workflow is not visible to the benchmarking system. It is more suitable for performance benchmarking than other alternatives since it usually has a smaller performance overhead.

Problem
How to observe the workflow execution behaviour during or after the execution of a test by relying on the engines APIs?

Forces

- Observing the behavior of workflow instances by interacting with engine-provided APIs produces timely information, BUT APIs usually differ for every engine. Therefore, the implementation of the benchmarking procedure requires more effort.

Solution

(1) Determine the APIs provided by the engine to access the workflow instances execution state;
(2) Execute the workflows;
(3) Use the API provided by the engine to query the deployment state of the workflow model, the current state, and the history of specific workflow instances and store these into a log;
(4) The benchmarking framework then reads the log traces after the execution of the test and compares them with expected ones.

Example – BenchFlow [Ferme et al. 2015] needs to access the execution state of the workflow instances to determine when the workflow engine completes the execution of the workflow. It relies on engines' provided APIs to verify the status of the started workflow instances after the load has been completely issued, and before collecting performance data.

Consequences

Benefits – Engine API-based Evaluation (P5.3) potentially provides a way to evaluate test results without a too strong influence on the system being benchmarked. The execution state of workflow instances and tests can be evaluated with a comparably little performance overhead.

Liabilities – The pattern may not necessarily be applicable, since it depends on the availability of an API for programmatic access to the engine functionality. Due to the current lack of standardized APIs, it also tends to complicate the development of the benchmarking procedure, since it usually requires to develop custom engine-specific code.

Known Uses

(1) This pattern is used in Betsy for BPMN conformance testing.
(2) BenchFlow [Ferme et al. 2015] queries the BPMN engines about their deployment status, prior to start issuing the load.
(3) BenchFlow [Ferme et al. 2015] queries the BPMN engines to monitor and verify the final states of the started workflow instances using engine APIs.

Relations

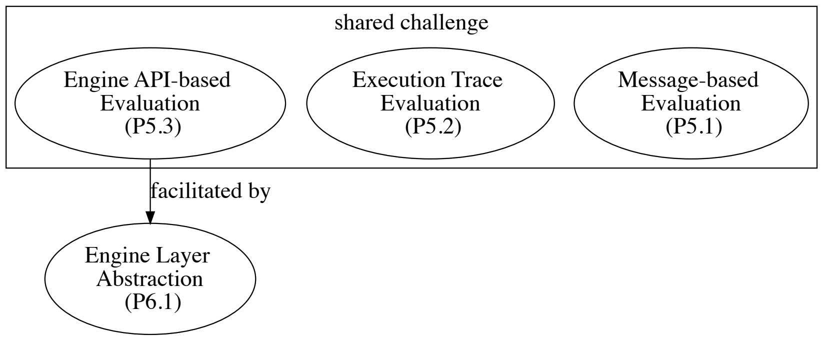

shared challenge
Engine API-based Evaluation (P5.3)
Execution Trace Evaluation (P5.2)
Message-based Evaluation (P5.1)
facilitated by
Engine Layer Abstraction (P6.1)

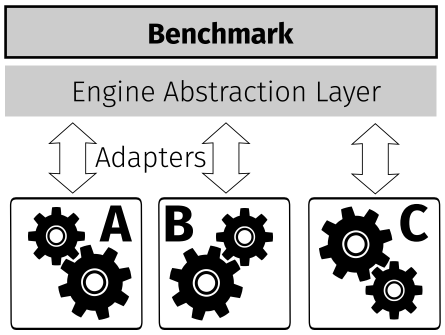
Fig. 11: Relations of P5.3 with other Patterns

Shared Challenge – This pattern is the most direct way of observing internal execution state available and an alternative to Message-based Evaluation (P5.1) and Execution Trace Evaluation (P5.2).
Facilitated by – The effort for applying this pattern is reduced with Engine Layer Abstraction (P6.1), because the interaction can be with the abstraction layer, instead of directly with the engine APIs.

4.6 Automatic Engine Interaction (C6) Patterns

Engine Layer Abstraction *(P6.1)*, and Failable Timed Action *(P6.2)* can be used to *automate the interaction with the engines (C6)*. Detailed Logs *(P6.4)* represents a way to obtain the required data from the engines for this interaction and Timeout Calibration *(P6.3)* shows a way to find a proper timeout for the interaction.

Engine Layer Abstraction (P6.1)



Summary
Since a benchmark is usually built for more than a single engine, an engine-independent layer that the benchmarking framework controls can help to streamline the operationalization of different engines.

Context
The purpose of benchmarking frameworks is to execute benchmarks on multiple products. APIs of engines might be vastly different.

Problem
How to interact with different engines uniformly?

Forces
- An engine-independent layer allows the benchmarking framework to interact with all engines uniformly, BUT this only is helpful for the operations supported by all engines and cannot take features or operations into account that are supported by a subset of the engines.
- An engine-independent layer allows to add a new engine without having to change the benchmarking procedure by implementing an adapter for that new engine, BUT implementing the adapter can require significant development effort.

Solution
(1) Define an abstract layer which a) converts engine independent artifacts to engine dependent ones and vice versa, and b) provides uniform methods to interact with each engine. This handles converting engine specific logs to engine independent log traces, engine installation, workflow deployment, workflow instance creation, and other engine specific operations such as how to behave after an abortion of a workflow;
(2) Implement an adapter for each engine

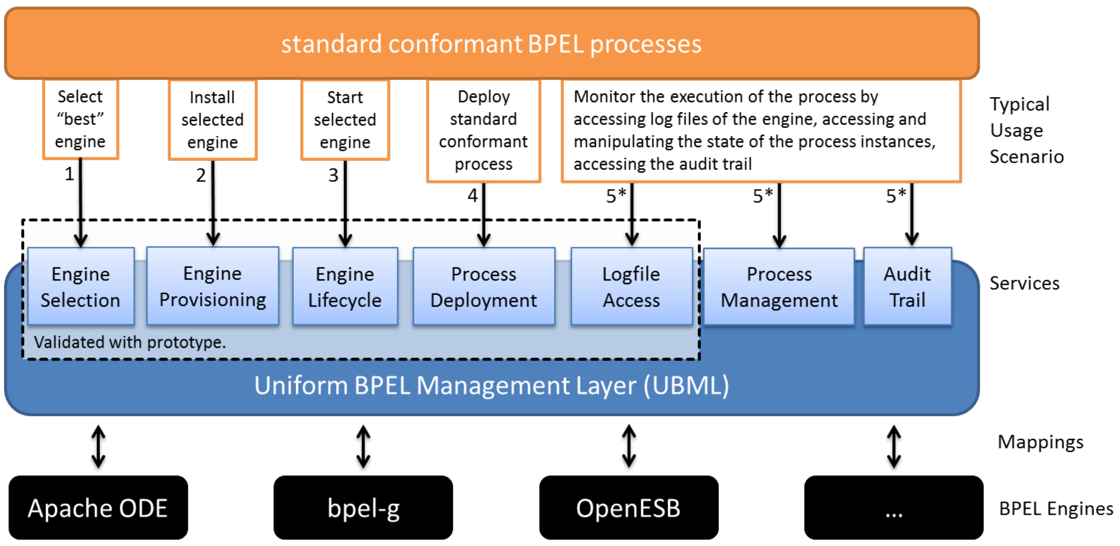

standard conformant BPEL processes
Select "best" engine
Install selected engine
Start selected engine
Deploy standard conformant process
Monitor the execution of the process by accessing log files of the engine, accessing and manipulating the state of the process instances, accessing the audit trail
Typical Usage Scenario
1
2
3
4
5*
5*
5*
Engine Selection
Engine Provisioning
Engine Lifecycle
Process Deployment
Logfile Access
Process Management
Audit Trail
Services
Validated with prototype.
Uniform BPEL Management Layer (UBML)
Mappings
Apache ODE
bpel-g
OpenESB
...
BPEL Engines

Fig. 12: Structure of the uniform BPEL management layer taken from [Harrer et al. 2014a]

Example – The uniform BPEL management layer (UBML) [Harrer et al. 2014a] contains a service which converts engine-independent BPEL processes with their corresponding WSDL and XSD files to engine-dependent and deployable packages, which can then be deployed. Each engine has its own format for its deployable packages and its own method on how to deploy their package. UBML captures all the different formats and methods and wraps them behind a uniform set of operations. The structure of UBML is depicted in Fig. 12.

Consequences

Benefits – Engines can be controlled in an abstract fashion independent of a concrete engine. The benchmarking procedure becomes more streamlined and easier to extend without modifications to the core procedure. It is easier to extend the benchmark with a new engine.

Liabilities – An adapter has to be developed per engine. The Engine Layer Abstraction (P6.1) may not contain all necessary operations and has to be circumvented for some operations.

Known Uses

(1) The pattern can be seen as a specialization of the *facade pattern* in object-orientation [Gamma et al. 1995b], and it is a widely used practice in systems integration [Grady 1994].
(2) The UBML [Harrer et al. 2014a] has been extracted from BPEL conformance benchmarking in Betsy. It is an engine independent layer to (un)install, start, and stop the engine as well as to deploy workflows and collect log files. The engine adapters of this layer heavily rely on Failable Timed Action (P6.2), Timeout Calibration (P6.3), and Detailed Logs (P6.4).
(3) A similar layer exists for BPMN conformance benchmarking in Betsy, as well.
(4) BenchFlow relies on an engine abstraction layer as well, by providing engine specific implementations for the APIs needed for deploying and starting process instances [Ferme et al. 2015].

Relations

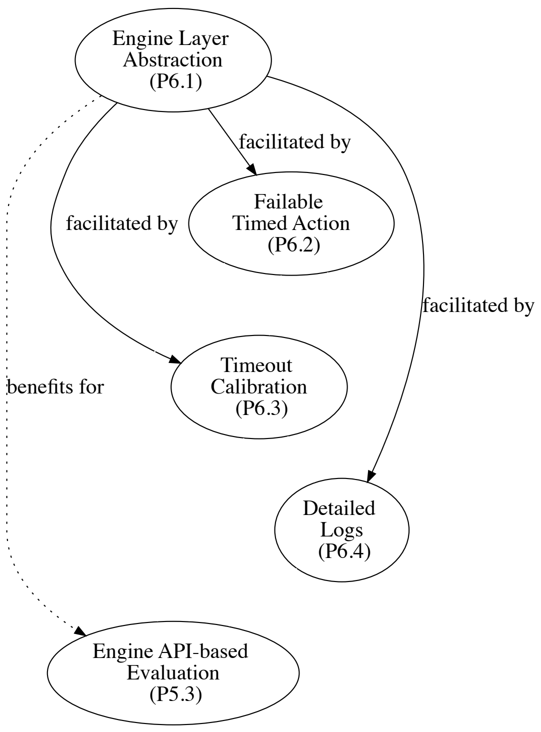
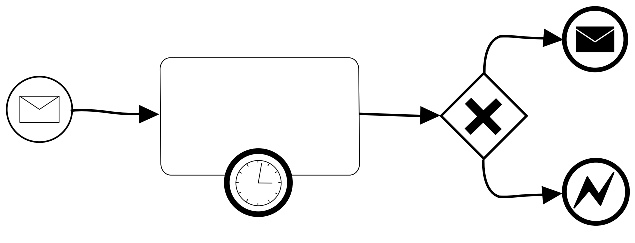

Engine Layer Abstraction (P6.1)
facilitated by
Failable Timed Action (P6.2)
facilitated by
Timeout Calibration (P6.3)
facilitated by
Detailed Logs (P6.4)
benefits for
Engine API-based Evaluation (P5.3)

Fig. 13: Relations of P6.1 with other Patterns

Facilitated by – The implementation of the pattern is facilitated by Failable Timed Action (P6.2), Timeout Calibration (P6.3), and Detailed Logs (P6.4).
Benefits for – The pattern supports the usage of Engine API-based Evaluation (P5.3).

Failable Timed Action (P6.2)



Summary
During a benchmark, engines may fail for various reasons and stop to make progress. To make sure that the overall benchmark continues nonetheless, it is necessary to restrict actions with a timeout in addition to waiting for the success or failure of the actions.

Context

Since execution failures can occur regardless of the type of benchmark, FAILABLE TIMED ACTION (P6.2) is relevant to performance and conformance benchmarking.

Problem

How to correctly complete the execution of a benchmark even when the benchmarked engines expose unexpected behaviour, e.g. indefinite waiting states?

Forces

- Setting a success, failure, and timeout condition for an action ensures that the benchmark will always progress, BUT the timeout can be set too low, causing the benchmark to proceed too early leading to flawed results, or the timeout can be set too high, causing the benchmark to take more time to execute.
- Handling timeouts for each action ensures that the benchmark will always progress, BUT the benchmarking framework requires additional logic to handle those timeouts.

Solution

(1) Each failable action needs a success, failure, and timeout condition.
(2) The test system executes a specified action.
(3) Then, it waits for a specific period during which success and failure conditions are checked every X milliseconds. The action fails if the time is exceeded or if a failure condition is met. It succeeds if the success condition is met within the specific period.

Example – The act of deploying a workflow often involves copying artifacts to a specific location on the file system, after which the engine deploys it automatically, and then evaluating success through log inspection. For instance, to deploy a BPEL workflow named “test.bpel” to Apache ODE, the deployment archive (a zip file containing the BPEL file and a deployment descriptor) must be placed in the folder named “processes” which is constantly monitored by Apache ODE for new deployment archives. If the workflow is deployed successfully, Apache ODE creates a marker file named “test.deployed” within the “processes” folder, and if the deployment fails, the log of Apache ODE contains the message “Deployment of test failed”.

Consequences

Benefits – The benchmark framework can detect failures on the side of workflow instances and it can make progress even when such failures occur.

Liabilities – Additional timeout handling results in more complex benchmark code.

Known Uses

(1) Betsy relies on FAILABLE TIMED ACTION (P6.2) for BPEL conformance benchmarks when interacting with the engines.
(2) it also relies on FAILABLE TIMED ACTION (P6.2) for BPMN conformance benchmarks when interacting with the engines.
(3) BenchFlow also relies on FAILABLE TIMED ACTION (P6.2) for issuing the load to the engines and checking whether the engine succeeds in handling the same.

Relations

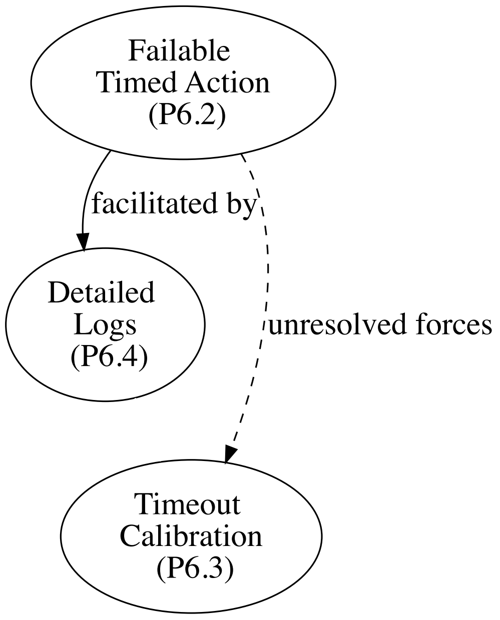
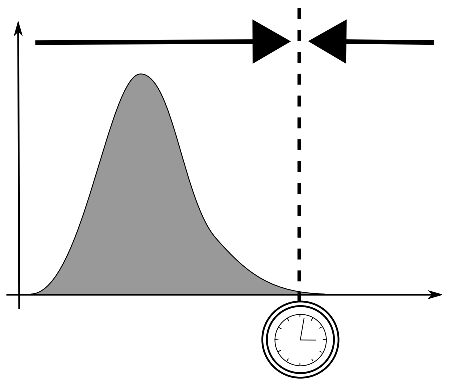


Fig. 14: Relations of P6.2 with other Patterns

Facilitated by – The pattern facilitates DETAILED LOGS (P6.4) because log messages can indicate success or failure conditions.
Unresolved Forces – TIMEOUT CALIBRATION (P6.3) is required by this pattern to make sure that timeouts are neither too high or too low.

Timeout Calibration (P6.3)



Summary

During a benchmark, usually many instances of FAILABLE TIMED ACTION (P6.2) are executed. Timeouts which are too long extend the benchmark duration unnecessarily and timeouts that are too short lead to flawed results. Timeout calibration helps preventing both cases.

Context

Since any type of benchmark is likely to depend on FAILABLE TIMED ACTION (P6.2), TIMEOUT CALIBRATION (P6.3) is relevant to performance and conformance benchmarking alike.

Problem

How to select an appropriate timeout for the various actions within a benchmark?

Forces

- Calibrated timeouts reduce the execution duration and the number of flaws in the result for subsequent benchmark runs, BUT calibrating the timeouts takes time and resources.
- Calibrating timeouts using a single workflow (e.g. APTITUDE TEST (P3.1)) saves time and resources, BUT the calibration may produce inappropriate timeouts for the full benchmark.
- Calibration provides appropriate timeouts within a specific environment and benchmark, BUT these timeouts may be inappropriate in another environment and benchmark.

Solution

(1) Before an actual machine is used for benchmarking, calibrate the timeouts by measuring the execution times of typical parts of a workflow.
(2) Use the calibrated timeouts for subsequent benchmark runs on that machine.

Example – Betsy calibrates its timeouts by measuring the time for each FAILABLE TIMED ACTION (P6.2) by conducting the APTITUDE TEST (P3.1) at least three times. To accommodate outliers, the timeout is the sum of the measured time plus a security range.

Consequences

Benefits – Calibrated timeouts reduce the execution duration caused by waiting too long and the number of flaws in the results caused by violated timeouts. Using a single workflow allows to quickly calibrate the timeouts on a specific machine.

Liabilities – The calibrated timeouts may not reduce execution time and the number of flaws in the results if the environment behaves differently, e.g. if the machine experiences additional load, or if the benchmark executes other actions than the ones that have been calibrated. The actions in full benchmarks may take longer than the actions during calibration, causing timeout violation during the benchmark, resulting in flawed results.

Known Uses

(1) Betsy uses this pattern for BPEL conformance benchmarking at several levels to limit the execution time that a benchmark consumes [Harrer et al. 2012]
(2) In the same fashion, Betsy uses this pattern to optimize the execution times of BPMN conformance benchmarks [Geiger et al. 2016b]
(3) In BenchFlow, timeouts for the response times were calibrated to make sure that engines had enough time to respond [Skouradaki et al. 2016].

Relations

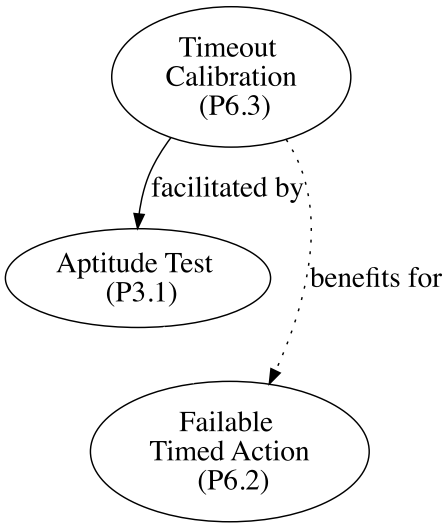
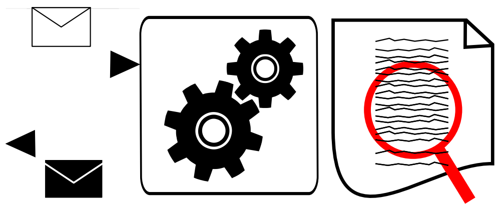


Fig. 15: Relations of P6.3 with other Patterns

Facilitated by – This pattern is facilitated by APTITUDE TEST (P3.1) because the simplest workflow possible is a good starting point for a resource efficient timeout calibration
Benefits for – The pattern benefits FAILABLE TIMED ACTION (P6.2) because it provides more meaningful timeout values.

Detailed Logs (P6.4)



Summary
To facilitate evaluating workflow engine behavior, enable detailed logging information.
Context
The behavior of workflow engines manifests in state changes during workflow execution. Those state changes are tracked by a workflow engine through the output of information into a log. This information can be used for evaluating the behavior of a workflow engine.
Problem
How to observe the workflow execution details (Workflow Execution Observation (C5)) and how to automate the interaction with the engines (Automatic Engine Interaction (C6))?

Forces

- For evaluating benchmark results, it has to be possible to infer the correctness of a workflow engine's execution. To support this, the goal of DETAILED LOGS (P6.4) is to extract as much information as possible out of the workflow engine, BUT the default workflow engine configuration typically does not provide enough information.

Solution

(1) Determine configuration options to configure logging;
(2) Determine possible log levels;
(3) Choose the log level with the highest verbosity;
(4) Then configure the workflow engine to use verbose logging.

In case other, non-verbose log levels are used, it might not be possible to observe everything that is important regarding the state of a workflow.

Example In the Camunda BPM engine, log levels can be configured using a properties file. The entry `org.camunda.bpm.engine.bpmn.behavior` is responsible for setting the log level of the engine [Camunda 2017]. Several log levels are offered, including `verbose` logging.

Consequences

Benefits – The execution state of an engine can be inspected after execution in its highest possible detail.
Liabilities – Increased logging level reduces workflow execution performance as more CPU and disk I/O is required.

Known Uses

(1) The pattern has been used for BPEL conformance benchmarking in Betsy [Harrer et al. 2012].
(2) It has also been used for BPMN conformance benchmarking in Betsy [Geiger et al. 2015].
(3) Benchflow uses the pattern for BPMN performance benchmarking [Skouradaki et al. 2016].
(4) From a more general-purpose point of view, the pattern has been used in software diagnosis [Yuan et al. 2012].

Relations

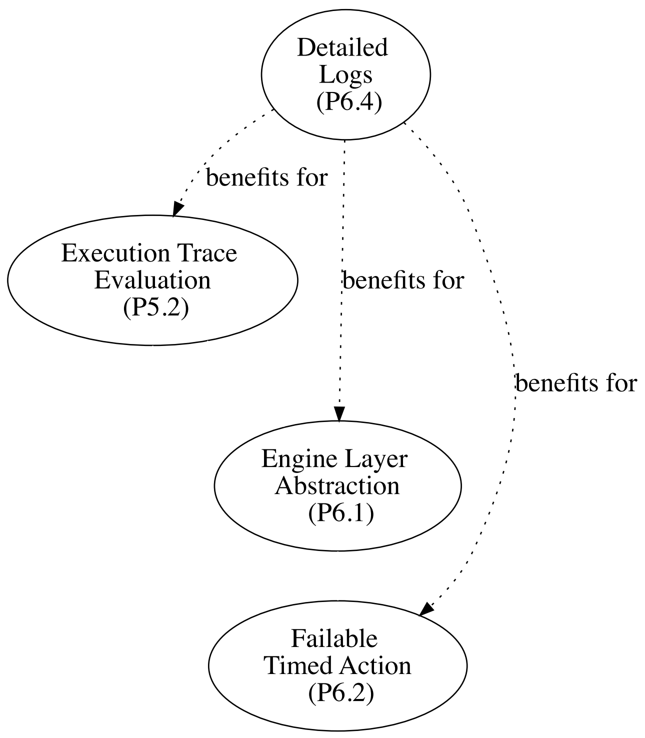

Detailed Logs (P6.4)
benefits for
Execution Trace Evaluation (P5.2)
benefits for
benefits for
Engine Layer Abstraction (P6.1)
Failable Timed Action (P6.2)

Fig. 16: Relations of P6.4 with other Patterns

Benefits for – EXECUTION TRACE EVALUATION (P5.2) can be implemented using this pattern. ENGINE LAYER ABSTRACTION (P6.1) builds on this pattern. FAILABLE TIMED ACTION (P6.2) may require this pattern to have enough information for deciding on action failure.

5. RELATIONS AMONG PATTERNS

The 15 patterns have various kinds of relationships to one another, which can be classified into the following four groups: *shared challenges*, *unresolved forces*, *facilitated by*, and *benefits for*. In Fig. 17 we present the complete pattern language presented in this work, and all the relations among all the patterns. In Fig. 18 to 21, these relationships are depicted. In particular, Fig. 18 shows the patterns that share the same challenge, Fig. 19 depicts the patterns that address unresolved forces of other patterns, Fig. 20 provides an overview over which patterns facilitate the implementation of others, and Fig. 21 outlines patterns that provide benefits for other patterns.

Each pattern is represented by an ellipse and each relationship by a directed edge. The only exception is that of the alternatives, as they are modeled through a rectangle in which every pattern is an alternative. The pattern relationships are represented in four separate figures to reduce the amount of overlapping edges, and make the representation of the relationships between the patterns easier to comprehend.

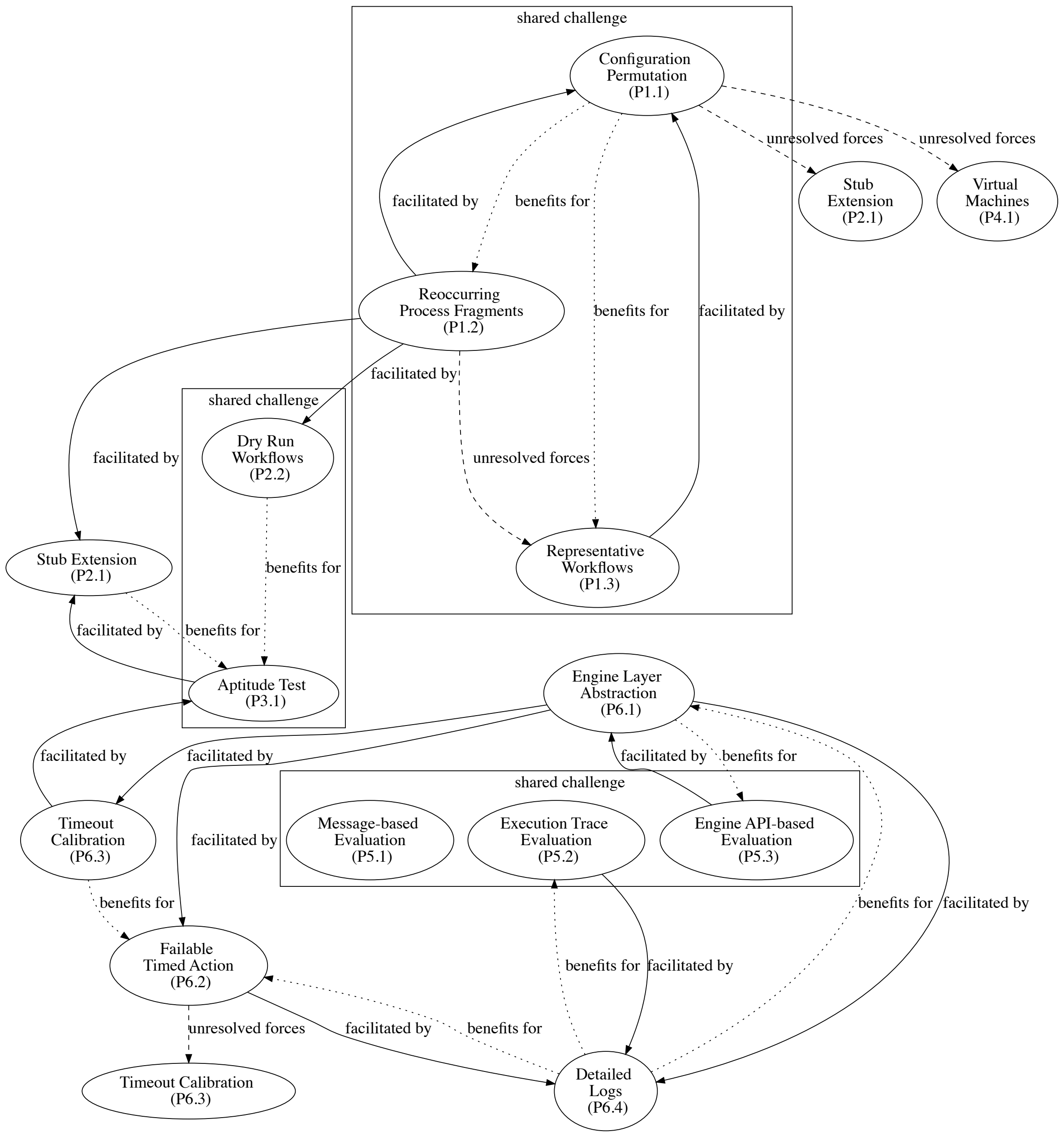


Fig. 17: Relationships Among Patterns

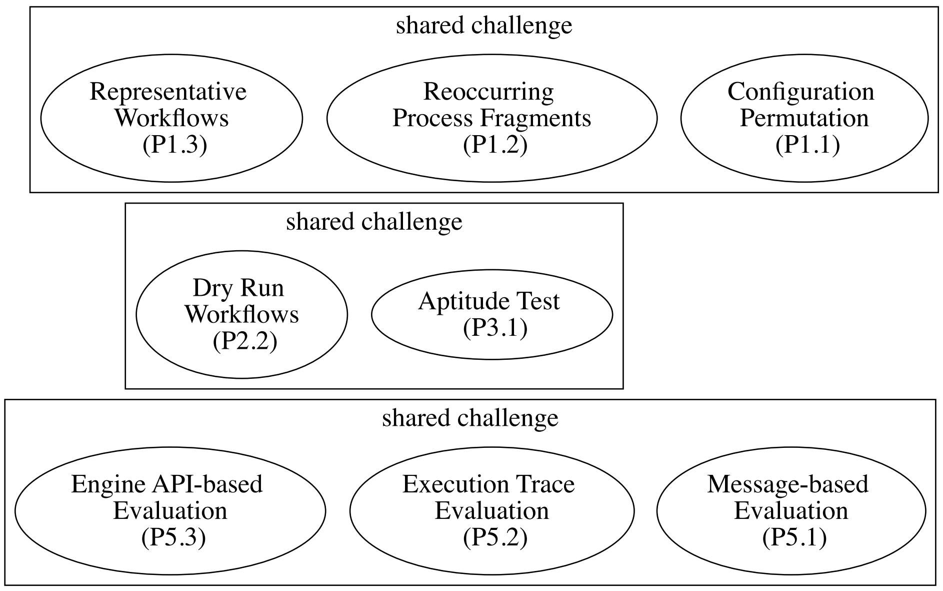

shared challenge
Representative Workflows (P1.3)
Reoccurring Process Fragments (P1.2)
Configuration Permutation (P1.1)
shared challenge
Dry Run Workflows (P2.2)
Aptitude Test (P3.1)
shared challenge
Engine API-based Evaluation (P5.3)
Execution Trace Evaluation (P5.2)
Message-based Evaluation (P5.1)

Fig. 18: “Shared Challenges” Relationships Among Patterns

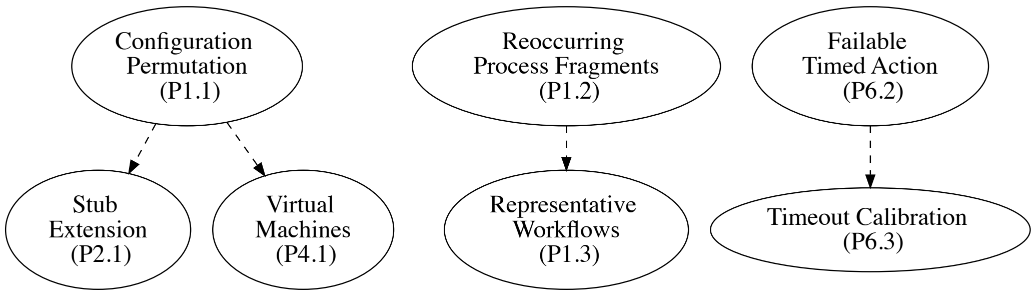

Configuration Permutation (P1.1)
Reoccurring Process Fragments (P1.2)
Failable Timed Action (P6.2)
Stub Extension (P2.1)
Virtual Machines (P4.1)
Representative Workflows (P1.3)
Timeout Calibration (P6.3)

Fig. 19: “Unresolved Forces” Relationships Among Patterns

The pattern relationships allowed us to uncover a number of additional insights. We can identify different patterns sequences, for example from patterns related to identify the tests (C1), to patterns related to validate the benchmarking procedure (C3): REOCCURRING PROCESS FRAGMENTS (P1.2) is facilitated by DRY RUN WORKFLOWS (P2.2) that has benefits for APTITUDE TEST (P3.1). Other patterns sequences are also present between patterns related to validate the benchmarking procedure (C3) and the ones related to automate the interaction with the engines (C6): ENGINE API-BASED EVALUATION (P5.3) is facilitated by ENGINE LAYER ABSTRACTION (P6.1), that is itself facilitated by DETAILED LOGS (P6.4), FAILABLE TIMED ACTION (P6.2)

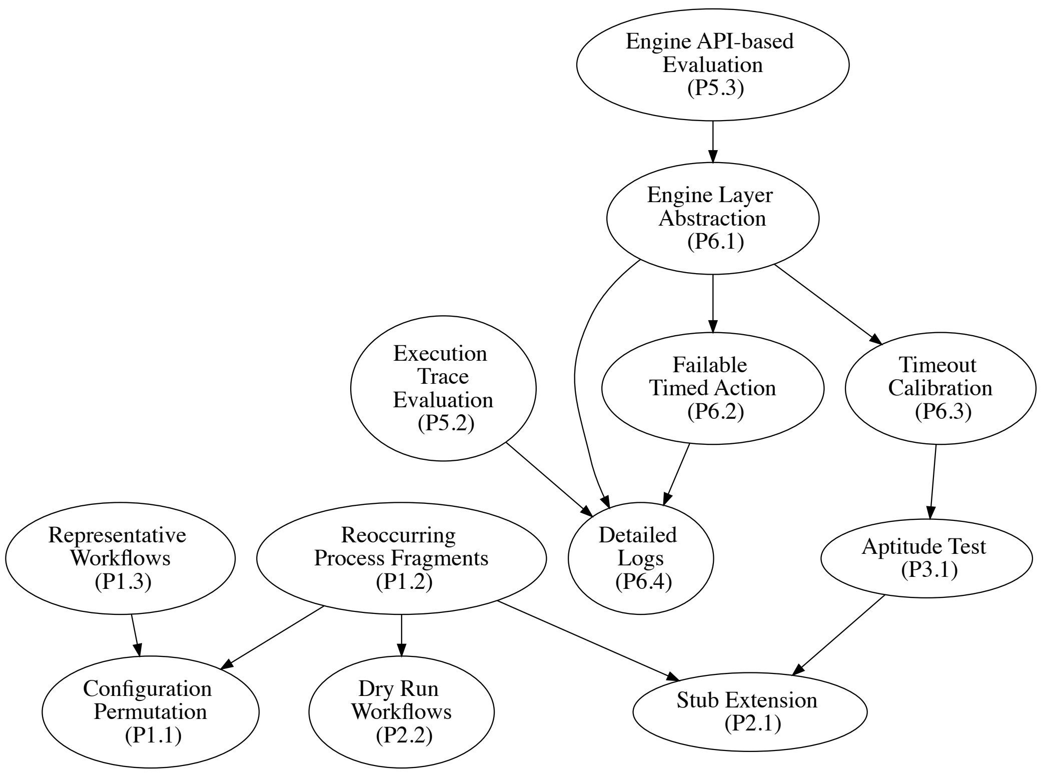

Engine API-based Evaluation (P5.3)
Engine Layer Abstraction (P6.1)
Execution Trace Evaluation (P5.2)
Failable Timed Action (P6.2)
Timeout Calibration (P6.3)
Representative Workflows (P1.3)
Reoccurring Process Fragments (P1.2)
Detailed Logs (P6.4)
Aptitude Test (P3.1)
Configuration Permutation (P1.1)
Dry Run Workflows (P2.2)
Stub Extension (P2.1)

Fig. 20: “Facilitated By” Relationships Among Patterns

and Timeout Calibration (P6.3). One pattern, namely Comparable Configuration (P3.2), does not have relationships with any other patterns, it stands alone. Another interesting aspect is that normally, there is only a single pattern for a specific problem, resulting in little choice between the patterns. Last, the two patterns Detailed Logs (P6.4) and Stub Extension (P2.1) can be viewed as foundational patterns of the language. Many other patterns rely on them, either directly and transitively. Together, they enable six other patterns, summing up to eight patterns which is more than half of the patterns language.

6. RELATED WORK

A number of related patterns have been discussed in the literature. Birukou [2010] presents an overview on methods for searching and selecting patterns. Unfortunately, most of the presented pattern repositories are no longer online and custom search engines for patterns are also no longer available. To find related patterns, we searched the databases of relevant publishers and indexing services and also the proceedings of the EuroPLoP and PLoP conferences from year 2000 onward.

In the context of testing, “Java Testing Patterns” [Thomas et al. 2004] provide guidelines for testing Java programs. The tests are mostly related to test object-oriented software and show the steps to test an application. Interestingly, none of the patterns are directly comparable to the presented patterns. Stobie [2001] presents patterns that describe how to formulate expected test results. Test data should be stored along with

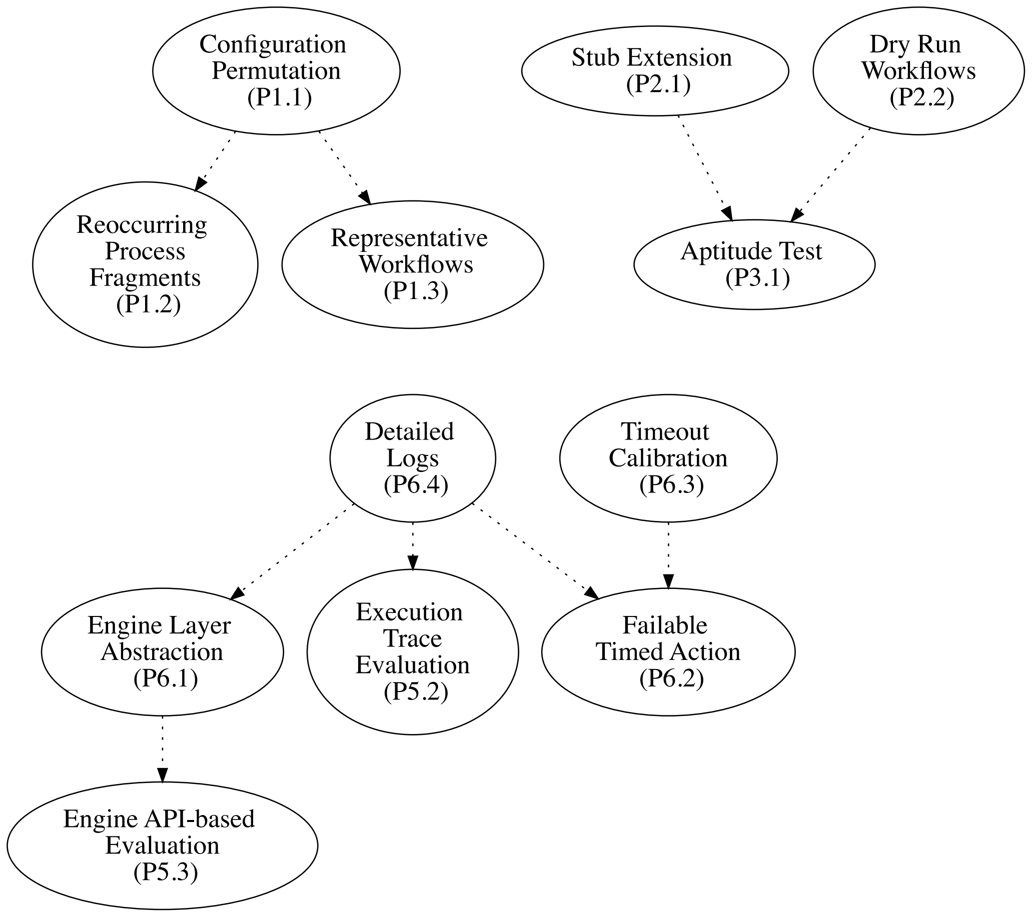

Configuration Permutation (P1.1)
Stub Extension (P2.1)
Dry Run Workflows (P2.2)
Reoccurring Process Fragments (P1.2)
Representative Workflows (P1.3)
Aptitude Test (P3.1)
Detailed Logs (P6.4)
Timeout Calibration (P6.3)
Engine Layer Abstraction (P6.1)
Execution Trace Evaluation (P5.2)
Failable Timed Action (P6.2)
Engine API-based Evaluation (P5.3)

Fig. 21: “Benefits For” Relationships Among Patterns

the test. This resembles the benchmarking tests as we apply them in our approaches, since we store all test artifacts in the same repository and not externally. For creation of the tests, we followed the “Check as you Go” pattern, which provides one assertion per test. Furthermore, Gassmann [2000] presents patterns for developing software tests in general. His work is centered around the concepts of JUnit and does not cover the black-box setting as it is done in this work. There is the pattern “WSDL-based testing of web service compositions” [Petrova-Antonova et al. 2015], which somehow details MESSAGE-BASED EVALUATION (P5.1): The message needs to be designed, assertions defined, and finally the test executed. VIRTUAL MACHINES (P4.1) is similar to the pattern “Virtual Machine Environment” [Syed and Fernandez 2016]. The VME pattern focuses on the general application of virtual machines, whereas we focus on benchmarking workflows. Oberortner et al. [2010] present patterns for measuring performance-related QoS properties. The aim is to measure the interaction between a client and a server and not to benchmark a single system as we do in our work.

A well-known set of patterns in the area of workflow management can be found in the workflow patterns, which exist in an initial [van der Aalst et al. 2003] and an extended set [Russel et al. 2006]. These publications define reoccurring control-flow structures in workflows and their popularity spawned work on patterns for other dimensions of workflows models. The same group of authors also envisaged workflow data patterns [Russell et al. 2005b], workflow resource patterns [Russell et al. 2005a], and workflow exception patterns [Russell et al. 2006]. Others groups contributed change patterns [Weber et al. 2008], time patterns [Lanz et al. 2010], process instantiation patterns [Decker and Mendling 2009], parallel computing patterns [Pautasso and Alonso 2006], and activity patterns [Thom et al. 2009]. The main difference between these pattern catalogs and the language we propose here lies in the focus of the patterns. The aforementioned articles describe patterns that target the language used to express workflows. They describe features that are commonly needed in practice and which, therefore, should be easily expressible in a workflow language. For instance, workflow languages should provide facilities to split the control-flow into parallel branches, as defined by the "parallel split" pattern [van der Aalst et al. 2003], or it should be possible to specify fixed dates at which an element of a workflow can be executed, as defined in the "fixed date element" pattern [Lanz et al. 2010]. That way, workflow patterns provide a means for comparing different workflow languages with each other and to evaluate which language is more suited for a certain domain, by supporting structures that are frequently needed in this domain. In contrast, we define patterns for building benchmarks for workflow engines, i.e., our patterns aim to compare the capabilities of runtime execution environments and not the capabilities of workflow languages. Since the runtime environments and workflow languages are necessarily intertwined, there also is a certain degree of overlap. Our overlap with workflow patterns is mainly captured in REOCCURRING PROCESS FRAGMENTS (P1.2), where we describe that workflow patterns from the articles listed above can be used as sources for process fragments. These fragments can then be used as test cases for benchmarking.

Picking up on the success of workflow patterns, similar patterns have been proposed in the area of service-oriented computing. These are the service interaction patterns [Barros et al. 2005], the correlation patterns [Barros et al. 2007] and the RESTful conversation patterns [Pautasso et al. 2016]. As discussed above, these pattern catalogs can be used as sources for REOCCURRING PROCESS FRAGMENTS (P1.2).

Lastly, workflow engines are one option for supporting and implementing enterprise integration scenarios. As such, enterprise integration patterns are related [Hohpe and Woolf 2004]. However, these patterns are rather centered on how to build an enterprise integration, whereas the patterns here are proposed for benchmarking one specific form of enterprise integration. In the same vein, process-oriented integration patterns do exist [Hentrich and Zdun 2006]. These patterns are more specific to integration scenarios using workflow engines, but as before are focused on implementing an integration between enterprises and not on benchmarking integration technology, which is our focus here.

7. CONCLUSION AND FUTURE WORK

In this paper, we presented a pattern language for workflow engine performance and conformance benchmarking. The language is structured around the four main elements involved in workflow engine benchmarking: the tests, the benchmarking procedure, the engine themselves, and the obtained results. A benchmark needs to address design challenges corresponding to each element, which can be solved by using one or more of the corresponding patterns. For each pattern, we provided a summary and describe its context, problem, forces, solution, consequences, known uses, and relations. The known uses of each pattern listed in the paper originate from both, our experience with the Betsy and BenchFlow projects, as well as from third party sources, such as [Rosinosky et al. 2016; Daniel et al. 2011]. Furthermore, we described the relationships among the patterns to draw a structure of the language as a whole.

The patterns collected in this paper provide a common vocabulary for benchmark authors as well as guidance regarding which alternative approaches should be selected. For example, there are three ways to observe the execution of the workflow: message-based, execution trace, and engine API-based evaluation.

As future work, this pattern language should be extended with all patterns from [Harrer et al. 2016] to provide a comprehensive reference for workflow engine benchmarking patterns. Moreover, the language could provide the foundation for standardization efforts for workflow engine benchmarking. Likewise, there is some potential in generalizing these specific patterns for benchmarking workflow engines to more generic patterns for software benchmarking or even to patterns for general benchmark design.

ACKNOWLEDGMENTS

We want to thank our shepherd, Allan Kelly, for the precious advice that helped us to improve the quality of our work, through continuous learning. He guided us in the process of improving the quality of the paper with his detailed feedback, and transparently sharing with us his constructive criticisms. We also want to thank all the participants of the writers workshop, for their valuable feedback that helped us to finalize the paper in its current state.

This work is partially funded by the "BenchFlow" project (SNF-DACH Grant Nr. 200021E-145062/1) and the "NEMAR" project (BMWi Grant Nr. 03ET4018).

REFERENCES

Christopher Alexander. 1978. A Pattern Language. (Aug. 1978).

Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. 2004. *Web Services.* Springer Nature. DOI:http://dx.doi.org/10.1007/978-3-662-10876-5

Marco Argenti. 2015. *Performance measurement of heterogeneous workflow engines.* Master's thesis. Università della Svizzera Italiana.

Xiaoying Bai, Wenli Dong, Wei-Tek Tsai, and Yinong Chen. 2005. WSDL-Based Automatic Test Case Generation for Web Services Testing. In *IEEE International Workshop on Service-Oriented System Engineering (SOSE'05).* IEEE. DOI:http://dx.doi.org/10.1109/sose.2005.43

Rabindra K. Barik, Rakesh K. Lenka, K. Rahul Rao, and Devam Ghose. 2016. Performance analysis of virtual machines and containers in cloud computing. In *2nd International Conference on Computing, Communication and Automation (ICCCA).* IEEE, 1204–1210. DOI:http://dx.doi.org/10.1109/ccaa.2016.7813925

Alistair P. Barros, Gero Decker, Marlon Dumas, and Franz Weber. 2007. Correlation Patterns in Service-Oriented Architectures. In *Proceedings of the 9th International Conference on Fundamental Approaches to Software Engineering (FASE'2007).* Springer Berlin Heidelberg, Braga, Portugal, 245–259. DOI:http://dx.doi.org/10.1007/978-3-540-71289-3_20

Alistair P. Barros, Marlon Dumas, and Arthur H. M. ter Hofstede. 2005. Service Interaction Patterns. In *3rd International Conference on Business Process Management.* Springer Berlin Heidelberg, Nancy, France, 302–318. DOI:http://dx.doi.org/10.1007/11538394_20

Domenico Bianculli, Walter Binder, and Mauro Luigi Drago. 2010a. Automated Performance Assessment for Service-oriented Middleware: A Case Study on BPEL Engines. In *Proceedings of the 19th international conference on World wide web (WWW'10).* ACM Press, 141–150. DOI:http://dx.doi.org/10.1145/1772690.1772706

Domenico Bianculli, Walter Binder, and Mauro Luigi Drago. 2010b. SOABench: Performance Evaluation of Service-oriented Middleware Made Easy. In *ICSE.* ACM, 301–302. DOI:http://dx.doi.org/10.1145/1810295.1810361

Aliaksandr Birukou. 2010. A survey of existing approaches for pattern search and selection. In *Proceedings of the 15th European Conference on Pattern Languages of Programs (EuroPLoP'10).* ACM Press. DOI:http://dx.doi.org/10.1145/2328909.2328912

Mustafa Bozkurt, Mark Harman, and Youssef Hassoun. 2012. Testing & Verification In Service-Oriented Architecture: A Survey. *Software Testing, Verificaton and Reliability* (2012). DOI:http://dx.doi.org/10.1002/stvr.1470

Camunda. 2017. Camunda Docs: Logging Categories: Process Engine. (2017). https://docs.camunda.org/manual/7.5/user-guide/logging/#process-engine

David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. 1997. The AETG System: An Approach to Testing Based on Combinatorial Design. *IEEE Transactions on Software Engineering* 23, 7 (jul 1997), 437–444. DOI:http://dx.doi.org/10.1109/32.605761

Florian Daniel, Giuseppe Pozzi, and Ye Zhang. 2011. *Workflow Engine Performance Evaluation by a Black-Box Approach.* Springer Berlin Heidelberg, Berlin, Heidelberg, 189–203. DOI:http://dx.doi.org/10.1007/978-3-642-25462-8_16

Gero Decker and Jan Mendling. 2009. Process Instantiation. *Data & Knowledge Engineering* 68, 9 (sep 2009), 777–792. DOI:http://dx.doi.org/10.1016/j.datak.2009.02.013

Jack Dongarra and Piotr Luszczek. 2011. *LINPACK Benchmark.* Springer US, Boston, MA, 1033–1036. DOI:http://dx.doi.org/10.1007/978-0-387-09766-4_155

Jozo Dujmović. 2010. Automatic Generation of Benchmark and Test Workloads. In *Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering (WOSP/SIPEW '10)*. ACM, New York, NY, USA, 263–274. DOI:http://dx.doi.org/10.1145/1712605.1712654

Vincenzo Ferme, Ana Ivanchikj, and Cesare Pautasso. 2015. A Framework for Benchmarking BPMN 2.0 Workflow Management Systems. In *BPM*. Springer. DOI:http://dx.doi.org/10.1007/978-3-319-23063-4_18

Vincenzo Ferme, Ana Ivanchikj, and Cesare Pautasso. 2016a. *Estimating the Cost for Executing Business Processes in the Cloud.* Springer International Publishing, Cham, 72–88. DOI:http://dx.doi.org/10.1007/978-3-319-45468-9_5

Vincenzo Ferme, Ana Ivanchikj, Cesare Pautasso, Marigianna Skouradaki, and Frank Leymann. 2016b. A Container-centric Methodology for Benchmarking Workflow Management Systems. In *CLOSER*. DOI:http://dx.doi.org/10.5220/0005908400740084

Vincenzo Ferme, Marigianna Skouradaki, Ana Ivanchikj, Cesare Pautasso, and Frank Leymann. 2017. Performance Comparison Between BPMN 2.0 Workflow Management Systems Versions. In *18th Enterprise, Business-Process and Information Systems Modeling (BPMDS)*. Springer International Publishing, 103–118. DOI:http://dx.doi.org/10.1007/978-3-319-59466-8_7

Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. 1995a. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Amsterdam.

Erich Gamma, John Vlissides, Richard Helm, and Ralph Johnson. 1995b. *Design patterns: Elements of reusable object-oriented software.* Addison-Wesley.

Venkatesh Ganti, Raghu Ramakrishnan, Johannes Gehrke, Allison Powell, and James French. 1999. Clustering large datasets in arbitrary metric spaces. In *15th International Conference On Data Engineering*. IEEE, 502–511.

Peter Gassmann. 2000. A Unit Testing Pattern Language. In *Proceedings of the 5th European Conference on Pattern Languages of Programms (EuroPLoP'2000)*.

Matthias Geiger, Simon Harrer, and Jörg Lenhard. 2016a. Process Engine Benchmarking with Betsy – Current Status and Future Directions. In *8th ZEUS Workshop (CEUR Workshop Proceedings)*, Vol. 1562. 37–44.

Matthias Geiger, Simon Harrer, Jörg Lenhard, Mathias Casar, Andreas Vorndran, and Guido Wirtz. 2015. BPMN Conformance in Open Source Engines. In *SOSE*. DOI:http://dx.doi.org/10.1109/SOSE.2015.22

Matthias Geiger, Simon Harrer, Jörg Lenhard, and Guido Wirtz. 2016b. On the Evolution of BPMN 2.0 Support and Implementation. In *SOSE*. 120–128. DOI:http://dx.doi.org/10.1109/sose.2016.39

Jeffrey O Grady. 1994. *System integration.* Vol. 5. CRC press.

Simon Harrer. 2014. Process Engine Selection Support. In *OTM 2014 Workshops*. LNCS, Vol. 8842. Springer, 18–22. DOI:http://dx.doi.org/10.1007/978-3-662-45550-0_3

Simon Harrer, Oliver Kopp, and Jörg Lenhard. 2016. Patterns for Workflow Engine Benchmarking. In *PEaCE in PATTWORLD: Joint Workshop on Performance and Conformance of Workflow Engines as well as Patterns and Pattern Languages for SOCC*. Vienna, Austria.

Simon Harrer, Jörg Lenhard, and Guido Wirtz. 2012. BPEL Conformance in Open Source Engines. In *SOCA*. IEEE, 237–244. DOI:http://dx.doi.org/10.1109/SOCA.2012.6449467

Simon Harrer, Jörg Lenhard, and Guido Wirtz. 2013. Open Source versus Proprietary Software in Service-Orientation: The Case of BPEL Engines. In *11th International Conference on Service Oriented Computing (ICSOC)*. Springer Berlin Heidelberg, Berlin, Germany, 99–113. DOI:http://dx.doi.org/10.1007/978-3-642-45005-1_8

Simon Harrer, Jörg Lenhard, Guido Wirtz, and Tammo van Lessen. 2014a. Towards Uniform BPEL Engine Management in the Cloud. In *Informatik 2014 (Lecture Notes in Informatics)*. GI e.V.

Simon Harrer, Cedric Röck, and Guido Wirtz. 2014b. Automated and Isolated Tests for Complex Middleware Products: The Case of BPEL Engines. In *ICSTW*. DOI:http://dx.doi.org/10.1109/ICSTW.2014.45

John A. Hartigan and Alexandra K. Wigdor (Eds.). 1989. *Fairness in Employment Testing: Validity, Generalization, Minority Issues, and the General Aptitude Test Battery.* National Academies Press, Washington, DC, USA. DOI:http://dx.doi.org/10.17226/1338

Carsten Hentrich and Uuwe Zdun. 2006. Patterns for Process-Oriented Integration in Service-Oriented Architectures. In *Proceedings of 11th European Conference on Pattern Languages of Programs (EuroPLoP'2006)*. Irsee, Germany, 1–45.

Gregor Hohpe and Bobby Woolf. 2004. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison Wesley, Amsterdam. ISBN: 0321200683.

Karl Huppler. 2009. The Art of Building a Good Benchmark. In *Performance Evaluation and Benchmarking*. Springer, 18–30. DOI:http://dx.doi.org/10.1007/978-3-642-10424-4_3

ISO/IEC. 2013. *ISO/IEC 19510:2013 – Information technology – Object Management Group Business Process Model and Notation*. v2.0.2.

Ana Ivanchikj. 2014. *Characterising Representative Models for BPMN 2.0 Workflow Engine Performance Evaluation*. Master's thesis. Università della Svizzera Italiana. https://thesis.bul.sbu.usi.ch/theses/1235-1314Ivanchikj/pdf?1412857872

Anil K Jain and Richard C Dubes. 1988. *Algorithms for clustering data*. Prentice-Hall, Inc.

Christian Kohls. 2010. The structure of patterns. In *17th Conference on Pattern Languages of Programs (PLoP'2010)*. ACM. DOI:http://dx.doi.org/10.1145/2493288.2493300

Christian Kohls. 2011. The structure of patterns – Part II – Qualities. In *18th Conference on Pattern Languages of Programs (PLoP'2011)*. ACM. DOI:http://dx.doi.org/10.1145/2578903.2601079

Andreas Lanz, Barbara Weber, and Manfred Reichert. 2010. Workflow Time Patterns for Process-Aware Information Systems. In *Enterprise, Business-Process, and Information Systems Modelling: 11th International Workshop BPMDS and 15th International Conference EMMSAD in conjunction with CAiSE*. Springer Berlin Heidelberg, Hammamet, Tunisia, 94–107. DOI:http://dx.doi.org/10.1007/978-3-642-13051-9_9

Kristian Bisgaard Lassen and Wil MP van der Aalst. 2009. Complexity metrics for Workflow nets. *Information and Software Technology* 51, 3 (mar 2009), 610–626. DOI:http://dx.doi.org/10.1016/j.infsof.2008.08.005

Jörg Lenhard, Andreas Schönberger, and Guido Wirtz. 2011. Edit Distance-Based Pattern Support Assessment of Orchestration Languages. In *19th International Conference on Cooperative Information Systems*. Springer Berlin Heidelberg, Hersonissos, Crete, Greece, 137–154. DOI:http://dx.doi.org/10.1007/978-3-642-25109-2_10

Zhongjie Li, Wei Sun, Zhong Bo Jiang, and Xin Zhan. 2005. BPEL4WS unit testing: framework and implementation. In *IEEE International Conference on Web Services*. IEEE, Orlando, FL, USA. DOI:http://dx.doi.org/10.1109/icws.2005.31

Gerard Meszaros and Jim Doble. 1998. A pattern language for pattern writing. *Pattern languages of program design* 3 (1998), 529–574.

Hafedh Mili, Guy Tremblay, Guitta Bou Jaoude, Éric Lefebvre, Lamia Elabed, and Ghizlane El Boussaidi. 2010. Business Process Modeling Languages: Sorting Through the Alphabet Soup. *ACM Comput. Surv.* 43, 1 (Nov. 2010), 4:1–4:56. DOI:http://dx.doi.org/10.1145/1824795.1824799

Ian Molyneaux. 2014. *The Art of Application Performance Testing*. O'Reilly Media, Inc.

OASIS. 2007. *Web Services Business Process Execution Language*. v2.0.

OASIS. 2017. *TOSCA Simple Profile in YAML Version 1.1*. Candidate OASIS Standard 01.

Ernst Oberortner, Uwe Zdun, and Schahram Dustdar. 2010. Patterns for measuring performance-related QoS properties in service-oriented systems. In *Proceedings of the 17th Conference on Pattern Languages of Programs (PLoP'10)*. ACM Press. DOI:http://dx.doi.org/10.1145/2493288.2493308

Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. 2008. Service-Oriented Computing: A Research Roadmap. *IJCIS* 17, 2 (jun 2008), 223–255. DOI:http://dx.doi.org/10.1142/s0218843008001816

Cesare Pautasso and Gustavo Alonso. 2006. Parallel Computing Patterns for Grid Workflows. In *Proceedings of the Workshop on Workflows in support for large-scale Science (WORKS06)*. IEEE, Paris, France. DOI:http://dx.doi.org/10.1109/works.2006.5282349

Cesare Pautasso, Ana Ivanchikj, and Silvia Schreier. 2016. A Pattern Language for RESTful Conversations. In *Proceedings of the 21st European Conference on Pattern Languages of Programs (EuroPLoP'2016)*. ACM, Kloster Irsee, Germany, 4:1–4:22. DOI:http://dx.doi.org/10.1145/3011784.3011788

Chris Peltz. 2003. Web Services Orchestration and Choreography. *IEEE Computer* 36, 10 (Oct. 2003), 46–52. DOI:http://dx.doi.org/10.1109/mc.2003.1236471

Dessislava Petrova-Antonova, Sylvia Ilieva, and Vera Stoyanova. 2015. A pattern for WSDL-based testing of web service compositions. In *Proceedings of the 20th European Conference on Pattern Languages of Programs (EuroPLoP'15)*. ACM Press. DOI:http://dx.doi.org/10.1145/2855321.2855324

Guillaume Rosinosky, Samir Youcef, and François Charoy. 2016. A Framework for BPMS Performance and Cost Evaluation on the Cloud. In *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE. DOI:http://dx.doi.org/10.1109/cloudcom.2016.0112

Anne Rozinat and Wil M. P. van der Aalst. 2008. Conformance Checking of Processes Based on Monitoring Real Behavior. *Information Systems* 33, 1 (mar 2008), 64–95. DOI:http://dx.doi.org/10.1016/j.is.2007.07.001

Nick Russel, Arthur H. M. ter Hofstede, Wil M. P. van der Aalst, and Nataliya Mulyar. 2006. *Workflow Control-Flow Patterns: A Revised View*. Queensland university of technology, eindhoven university.

Nick Russell, A. H. M. ter Hofstede, and David Edmond. 2005a. Workflow Resource Patterns: Identification, Representation and Tool Support. In *Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSE05)*. Springer, Porto, Portugal, 216–232. DOI:http://dx.doi.org/10.1007/11431855_16

Nick Russell, A. H. M. ter Hofstede, David Edmond, and W. M. P. van der Aalst. 2005b. Workflow Data Patterns: Identification, Representation and Tool Support. In *Proceedings of the 24th International Conference on Conceptual Modeling (ER2005)*. Springer, Klagenfurt, Austria, 353–368. DOI:http://dx.doi.org/10.1007/11568322_23

Nick Russell, Wil M. P. van der Aalst, and Arthur H. M. ter Hofstede. 2006. Workflow Exception Patterns. In *Proceedings of the 18th Conference on Advanced Information Systems Engineering (CAiSE06)*. Springer Berlin Heidelberg, Luxembourg, Luxembourg, 288–302. DOI:http://dx.doi.org/10.1007/11767138_20

Richard F Schmidt. 2013. *Software engineering: Architecture-driven software development*. Newnes.

Marigianna Skouradaki, Vincenzo Ferme, Cesare Pautasso, Frank Leymann, and Andr'e van Hoorn. 2016. Micro-Benchmarking BPMN 2.0 Workflow Management Systems with Workflow Patterns. In *CAiSE*. Springer, 67–82. DOI:http://dx.doi.org/10.1007/978-3-319-39696-5_5

Keith Stobie. 2001. Test Result Handling. In *8th Conference on Pattern Languages of Programs (PLoP'2001)*.

Madiha H. Syed and Eduardo B. Fernandez. 2016. A Pattern for a Virtual Machine Environment. In *Proceedings of the 23rd Conference on Pattern Languages of Programs (PLoP'16)*.

Lucinéia Heloisa Thom, Manfred Reichert, and Cirano Iochpe. 2009. Activity Patterns in Process-aware Information Systems: Basic Concepts and Empirical Evidence. *International Journal of Business Process Integration and Management (IJBPIM)* 4, 2 (2009), 93–110. DOI:http://dx.doi.org/10.1504/ijbpim.2009.027778

Jon Thomas, Matthew Young, Kyle Brown, and Andrew Glover. 2004. *Java Testing Patterns*. Wiley.

Jóakim v. Kistowski, Jeremy A. Arnold, Karl Huppler, Klaus-Dieter Lange, John L. Henning, and Paul Cao. 2015. How to Build a Benchmark. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE '15)*. ACM, New York, NY, USA, 333–336. DOI:http://dx.doi.org/10.1145/2668930.2688819

Wil M. P. van der Aalst. 2013. Business Process Management: A Comprehensive Survey. *ISRN Software Engineering* (2013), 1–37. DOI:http://dx.doi.org/10.1155/2013/507984

Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. 2003. Workflow Patterns. *Distributed and Parallel Databases* 14, 1 (July 2003), 5–51. DOI:http://dx.doi.org/10.1023/A:1022883727209

Irene Vanderfeesten, Hajo A Reijers, and Wil M.P. van der Aalst. 2008. Evaluating workflow process designs using cohesion and coupling metrics. *Computers in industry* 59, 5 (may 2008), 420–437. DOI:http://dx.doi.org/10.1016/j.compind.2007.12.007

Barbara Weber, Stefanie Rinderle, and Manfred Reichert. 2008. Change Patterns and Change Support Features – Enhancing Flexibility in Process-Aware Information Systems. *Data and Knowledge Engineering, Elsevier* 66, 3 (July 2008), 438–466. DOI:http://dx.doi.org/10.1016/j.datak.2008.05.001

WfMC. 1995. *The Workflow Reference Model*. v1.1.

Petia Wohed, Marlon Dumas, Arthur H. M. Ter Hofstede, and Nick Russell. 2006. *Pattern-based Analysis of BPMN – An extensive evaluation of the Control-flow, the Data and the Resource Perspectives (revised version)*. BPM Center Report BPM-06-17.

Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. 2012. Improving Software Diagnosability via Log Enhancement. *ACM Transactions on Computer Systems* 30, 1 (2012). DOI:http://dx.doi.org/10.1145/2248487.1950369